Best
Available
Copy

AD-763 673

MODEL BASED (INTERMEDIATE-LEVEL) COMPUTER VISION

STANFORD UNIVERSITY

PREPARED FOR
ADVANCED RESEARCH PROJECTS AGENCY

MAY 1973

Distributed By:

NTIS

National Technical Information Service
U. S. DEPARTMENT OF COMMERCE

STANFORD ARTIFICIAL INTELLIGENCE
MEMO AIM-201

STAN-CS-73-366

AD 763673

MODEL BASED (INTERMEDIATE-LEVEL) COMPUTER VISION

BY

GUNNAR RUTGER GRAPE

SUPPORTED BY
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

MAY 1973

DDC
RECEIVED
JUL 26 1973
B

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)

Stanford University
Dept. of Computer Science
Stanford, California 94305

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3. REPORT TITLE

MODEL BASED (INTERMEDIATE-LEVEL) COMPUTER VISION

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

technical report, May 1973

5. AUTHOR(S) (First name, middle initial, last name)

Gunnar R. Grape

6. REPORT DATE

May 1973

7a. TOTAL NO. OF PAGES

~~approx.~~ ~~256~~ 266

7b. NO. OF REFS

8a. CONTRACT OR GRANT NO

ARPA-SD-183

b. PROJECT NO

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

STAN-CS-73-366

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

AIM201

10. DISTRIBUTION STATEMENT

Releasable without limitations on dissemination.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

A system for computer vision is presented, which is based on two-dimensional prototypes, and which uses a hierarchy of features for mapping purposes.

More specifically, we are dealing with scenes composed of planar faced, convex objects. Extensions to the general planar faced case are discussed.

The visual input is provided by a TV-camera, and the problem is to interpret that input by computer, as a projection of a three-dimensional scene.

The digitized picture is first scanned for significant intensity gradients (called edges), which are likely to appear at region-and object junctions. The two-dimensional scene-representation given by the totality of such intensity discontinuities (that word used somewhat inexactly) is referred to in the sequel as the "edge-drawing", and constitutes the input to the vision system presented here.

The system proposed and demonstrated in this paper utilizes perspectively consistent two-dimensional models (prototypes) of views of three-dimensional objects, and interpretations of scene-representations are based on the establishment of mapping relationships from conglomerates of scene-elements (line-constellations) to prototype templates. The prototypes are learned by the program through analysis of - and generalization on - ideal instances.

The system works better than any sequential (or other) system presented so far. It should be well suited to the context of a complete vision system, using depth, occlusion, support relations, etc. The general case of irregularly shaped, planar faced objects, including concave ones, would necessitate such an extended context.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6801

Unclassified

Security Classification

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT

DD FORM 1 NOV .. 1473 (BACK)
(PAGE 3)

Security Classification

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-201

APRIL 1973

COMPUTER SCIENCE DEPARTMENT
REPORT CS-366

MODEL BASED (INTERMEDIATE-LEVEL) COMPUTER VISION

by

Gunnar Rutger Grape

ABSTRACT: A system for computer vision is presented, which is based on two-dimensional prototypes, and which uses a hierarchy of features for mapping purposes.

More specifically, we are dealing with scenes composed of planar faced, convex objects. Extensions to the general planar faced case are discussed.

The visual input is provided by a TV-camera, and the problem is to interret that input by computer, as a projection of a three-dimensional scene.

The digitized picture is first scanned for significant intensity gradients (called edges), which are likely to appear at region- and object junctions. The two-dimensional scene-representation given by the totality of such intensity discontinuities (that word used somewhat inexactly) is referred to in the sequel as the "edge-drawing", and constitutes the input to the vision system presented here.

The system proposed and demonstrated in this paper utilizes perspectively consistent two-dimensional models (prototypes) of views of three-dimensional objects, and interpretations of scene-representations are based on the establishment of mapping relationships from conglomerates of scene-elements (line-constellations) to prototype templates. The prototypes are learned by the program through analysis of - and generalization on - ideal instances.
The system works better than any sequential (or other) system presented so far. It should be well suited to the context of a complete vision system, using depth, occlusion, support relations, etc. The general case of irregularly shaped, planar faced objects, including concave ones, would necessitate such an extended context.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract No. SD-183.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Project Agency or the U. S. Government.

ABSTRACT

A system for computer vision is presented, which is based on two-dimensional prototypes, and which uses a hierarchy of features for mapping purposes.

More specifically, we are dealing with scenes composed of planar faced, convex objects. Extensions to the general planar faced case are discussed.

The visual input is provided by a TV-camera, and the problem is to interpret that input by computer, as a projection of a three-dimensional scene.

In this case the digitized picture is first scanned for significant intensity gradients (called edges), which are likely to appear at region- and object junctions. The two-dimensional scene-representation given by the totality of such intensity discontinuities (that word used somewhat inexactly) is referred to in the sequel as the "edge-drawing", and constitutes the input to the vision system presented here.

If edge-drawings were perfect, the task of interpreting them, that is of determining the composition of the scene (in terms of partaking objects), would not be an excessively hard one. A rather simple scheme of sequential abstractions would work adequately, obtaining successively higher levels of abstraction (information compression), in some order like: Edges - Lines - Vertices - Regions - Bodies - Scene.

Unfortunately, edge-drawings are very seldom anywhere near perfect, due to effects like shadows, glare, reflections, insufficient intensity gradients between regions, hardware imperfections, digitization errors, etc. The sequential approach, therefore, does not work adequately in practice. The need for more global information (even at low levels of abstraction) has become more evident with every effort put into the development of sequential vision schemes.

The system proposed and demonstrated in this paper utilizes perspectively consistent two-dimensional models (prototypes) of views of three-dimensional objects, and interpretations of scene-representations are based on the establishment of mapping relationships from conglomerates of scene-elements (line-constellations) to prototype templates. The prototypes are learned by the program through analysis of - and generalization on - ideal instances.

A small hierarchy of features (specific line- and vertex constellations) is used in providing entry-points (keys) into such mappings, since an exhaustive search is out of the question (for reasons of combinatorics).

Features are also used during the process of mapping scene-elements onto a prototype, serving now as guides and templates.

This system is intermediate-level in the sense that it does not work on the basis of the original TV-image (but on information abstracted from it), and that it does not determine (or use) spatial dimensions, positions, or relationships of the objects in a given scene.

Its place in an extended three-dimensional system is discussed, as are some possible aspects of such a system.

The results obtained are quite good, using scenes of realistic complexity and with many examples of different kinds of imperfections in the initial data.

In conclusion, the system works better than any sequential (or other) system presented so far. It should be well suited to the context of a complete vision system, using depth, occlusion, support relations, etc. The general case of irregularly shaped, planar faced objects, including concave ones, would necessitate such an extended context.

ACKNOWLEDGMENTS

I wish to express my thanks to Professor Jerome Feldman for his invaluable help as my thesis adviser, and to Professor Cordell Green and Dr. Thomas Binford for serving on the reading committee.

I gratefully acknowledge the helpfulness of my fellow workers, in particular Dr. Manfred Hueckel, Dr. Richard Paul, and Mr. Karl Pingle.

The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

FIGURE

1.0 INTRODUCTION

In the context of the present paper, the term "computer vision" is restricted in scope to a world of planar-faced solids, notably parallelepipeds, wedges, and other simple objects that may be expected to be useful in a "Hand-Eye" context, for instance as building blocks.

This paper deals more or less exclusively with vision at an intermediate level, viz. our "input" is an array of "brightness-discontinuity points" (over a digitized TV-raster representing our view of the scene). Our "output" is a formalized interpretation of that information as a two-dimensional representation of a three-dimensional scene.

Statement of the problem:
On a table is a collection of blocks. "Looking" at that scene is a TV-camera, which is linked to the computer, so that the latter may obtain a digitized raster of the image. The problem, of course, is to program the computer so as to enable it to output an interpretation of that TV-image, in terms of the nature and relationships of the objects in the scene. This interpretation may then be the final product in itself, or it may be used by other programs for purposes of manipulating the objects in the scene.

At this early point I suggest that the reader take a look at at some of the examples in Section 11, in order to get a more concrete idea of what this is all about.

1.0

This paper first touches on some related efforts toward solving the vision problem, and on the pros and cons of sequential (abstraction) versus model driven schemes. Its main body presents a system based on prototypes and features, which uses global knowledge and goal-direction to a higher extent than has previously been tried.

The basic lay-out of the present paper is the following:

Immediately after this introduction you will find a list of commonly used terms and abbreviations (Section 2). I recommend a quick scan through that list before reading the rest of this presentation, but the main use of the list should be for easy reference whenever unfamiliar terms are encountered throughout this paper.

Section 3 deals with previous and related efforts in this area of interest, and contains a discussion of the difficulties inherent in "sequential-abstraction" methods, contrasting that approach with "global-knowledge" schemes, particularly model-based ones.

Section 4 through Section 12, the main part of this presentation, describe a model-based, feature-driven, intermediate-level vision system. Examples of system performance are provided, as well as a discussion of future possibilities, and the usual conclusions. There is also an appendix containing a description of the general data-structure.

The main part of the thesis is presented according to the following plan:

First an overview of general considerations and basic strategies, Section 4. Then a thorough discussion of the feature hierarchy (definitions - properties - utilization), Section 5. Section 6 then introduces the prototype concept, in like manner and considerable detail.

Having thus established the conceptual machinery, we then embark on a description of the process of interpreting the scene.

Preprocessing is dealt with in Section 7. The parsing process (which utilizes the prototype matching program in interpreting the scene) is presented in Section 8. The matching process is given in the following section, which is rather technical and relies on a hierarchy of block-diagrams for the presentation of the flow of process.

I have found it difficult to give a transparent account of the matching program, and I ask the reader's indulgence, should she/he find the presentation hard to absorb at first glance.

Section 10 discusses a possible object completion phase. This is followed by examples of system performance (Section 11), and discussions of future possibilities (Section 12). The last three sections are (in order of appearance): Conclusions, Appendix, and Bibliography.

For subdivisions of the above, see table of contents.

2.0 COMMON DEFINITIONS AND ABBREVIATIONS

BARE (vertex): A vertex consisting of a single line-end, possibly before insertions of extra lines.

BASE-LINE: See PARENT LINE.

CCW.: Counter clockwise.

CF: Compound feature (Section 5).

COMPLEXITY: The number of lines involved (in a vertex, a feature, or a prototype).

CONNECTED: See SIMPLY CONNECTED.

CONSTELLATION: Usually the group of lines referenced by a vertex or a feature (should be clear in each context).

CONVEX (object): An object where any line connecting two points of the surface lies entirely inside the object or on its surface.

CUT: A line (in the drawing) chopping a small piece off another line, in the formation of a vertex.

EQUALITY CLASS: Collective term for length class and parallelity class.

Preceding page blank

EQUIVALENCE CLASS: Pertaining to LF:s in a prototype (Subsection 6.3).

ILV-SYSTEM: Intermediate-Level Vision System.

LENGTH CLASS: Pertaining to lines in a prototype (Subsection 6.4).

LF: Line-feature (Section 5).

MODEL (and derivatives): Alluding to a concrete pattern (prototype) for matching, or an abstract, driving concept, such as the idea of "object" or "well-shaped region". Connected with the use of global contexts (cf. Subsection 3.3). Sometimes "model" is used interchangeably with "prototype".

OBJECT: Usually a physical entity, such as a block on the table. Also used for the perceptual entity of an internal object representation.

ORBIT: Orbiting a vertex means cycling around it in a ccw. direction, visiting the lines one by one, from some given starting line.

ORBITAL DISTANCE: The number of lines from (excluding) a given line up to (and including) another, in a ccw. direction around a vertex.

PARALLELITY CLASS: Pertaining to lines in a prototype (Subsection 6.4).

PARENT LINE(S) (of a feature): The line (or lines) which is (are) partaking fully (i.e. with both ends) in the feature.

PARTIAL: Short for partially matched object.

P-LINE: Prototype line.

RAY: A line-segment with only one end and a direction given or currently referenced.

SCENE: A collection of real-world objects, or the internal representation thereof.

SEQUENTIAL (and derivatives): Usually referring to the idea of "sequential abstractions" (cf. MODEL and Subsection 3.3). Not used as opposite of "parallel" (processing).

SIMPLY CONNECTED: Two vertices are simply connected iff they have a line in common, two non-parallel lines iff they share a vertex, two parallel lines iff they have a connecting line.

SUCCESSOR LINE: The line following a given line, in the orbit of a vertex.

TOPOLOGY: Besides its usual meaning, it is sometimes used in conjunction with features, namely with their normal context in mind (as parts of complete topologies).

TRIHEDRAL: Sometimes short for TRIHEDRAL OBJECT.

TRIHEDRAL OBJECT: A planar faced object with TRIHEDRAL VERTICES only.

TRIHEDRAL VERTEX: A vertex where exactly three surfaces meet.

3.0 APPROACHES TO THE VISION PROBLEM

In most fields of science the established pattern of research has been the selection and investigation of subproblems, rather than broad frontal attacks on complex systems. Of course, practicality oftentimes prompts such policies, and mostly the results achieved are relevant to the understanding or function of the whole.

I am not quite certain whether research in computer vision fits into such a pattern.

During the last decade, many man-years have been spent on investigating problems conceived as relevant parts of some nebulous whole. To the extent that we have gathered understanding of the difficulties inherent in vision, the results have certainly been relevant. Whether they are applicable in the context of future, complete vision systems, is a different consideration.

Research in this field has been more or less confined to an idealized world of objects whose surfaces are all planar. The rationale behind this is twofold. Such objects are comparatively easy to represent in a computer, and many every-day manipulatory tasks, interesting from the standpoint of Artificial Intelligence, involve such cojects.

The implicit assumption has been that from this kind of first approximation to computer vision we should be able to build more generally applicable systems.

Such assumptions are dangerous, in my opinion. The confinement to planar faced objects has invited all kinds of klugery and special-case analysis that is completely irrelevant to perception of more general objects.

Many people have elected to further limit the scope of their research to the segmentation of ideal line-representations of scenes, in terms of their constituent object-interpretations. While that subproblem is by no means trivial, and certainly elucidating in its own right, it would seem not immediately relevant to the intensely practical realities of computer vision, even in the restricted context.

The seeming simplicity of the subproblem (dealing with planar faced objects) has seduced us into attempting solutions with limited machinery, using restrictive assumptions and special-case heuristics. I think this is unfortunate, but maybe a "necessary" way to develop this young field of research.

Vision is a hard problem. I guess we have all learned from the lack of spectacular results so far.

This section deals with related history in computer vision, describes the mad quest for the Perfect Line-drawing (alias Pimpernel), and discusses hierarchical (local decision) versus model driven schemes.

3.1 BRIEF PERSPECTIVE ON RELATED EFFORTS

" . . . Is He in Heaven .(?). Is He in Hell .(?).
That damned elusive Pimpernel?!"

[Roberts 1963]:

For current purposes the history of computer vision starts with Roberts. His work covered the complete spectrum, from camera output to three-dimensional interpretation, and is in that sense a unique effort. Most other work in computer vision, so far, has dealt with subsystems. But even Roberts paid scant attention to the pre-processing stages of his system, concentrating on the aspects of handling representations of three-dimensional objects.

Using a facsimile scanner on a photograph of the scene to provide picture input, he then deploys some fairly simple heuristics to abstract a connected line-drawing from the original raster. The program subsequently finds all well-shaped regions, and attempts to match constellations of such regions with similar constellations recorded for the three-dimensional models. This is performed in a series of steps, each one performed when the previous one yields no results, and each one requiring less information than the previous one:

1. Using regions around a vertex.
2. Using regions surrounding a line.
3. Using a region and a third line from one vertex.
4. Using a three-line vertex.

Note that step 4 signifies a liberation from the requirement for well-shaped regions, but since 4 points are enough to determine a perfect partial projection of any of his models, this easily leads to forbidding combinatorics and nonsensical interpretations in non-trivial scenes.

A fixed set of 3 models (parallelepiped, wedge, hexagonal prism) is used and, given the key match, the picture and model points are "cycled around" in order to "line up the order of the polygons". If the orders can be matched, a list of equivalent point-pairs is created, the transformation from the 3D model to the scene representation is computed, as well as the error of fit. If the match is acceptable the lines belonging to the object projection are removed from the scene, and the process iterates.

The treatment of composite objects, or rather, the interpretation of complex objects as conglomerates of instances of his simple models, is of particular interest. When a picture polygon is divided during the process of back-projecting an object, lines inside that region - and belonging to that object - are inserted, and that fact is remembered so that a linkage of the parts of a composite object may be obtained. Such objects may then be back-projected in any position and under any rotation.

The models are not fixed as to size, so that any right-angle parallelepiped would match his "cube"-model, for instance. They are fixed as to skew, however.

To sum up, Roberts work constitutes an important initial effort. His program worked on very simple scenes under ideal conditions. The preprocessing heuristics are not sophisticated enough to handle complex scenes, and the matching program seems highly dependent on perfect line-drawings. The program also seems dependent on the fixed set of models, so that the incorporation of a new model would require program changes. This drawback is somewhat offset by allowing for composite objects. The treatment of such objects, and the three-dimensional manipulations, are particularly interesting.

Quoting Roberts: "The biggest benefit of this investigation, however, is an increased understanding of the possible processes of visual perception."

[Guzman 1968]:

The major contribution of Guzman was the demonstration that - for quite complex scenes - assuming essentially perfect line-drawings without shadows or other irregularities - one may very often infer interpretations in terms of projections of three-dimensional objects (body segmentation), using a rather limited set of chiefly local heuristics based on the properties of the vertices in the scenes.

Using such heuristics, weak or strong links (depending on the vertices in question) may be established between regions across lines. Regions are subsequently grouped together into "nuclei" according to the nature of such links, and the final nuclei constitute the body interpretations.

Little global context is used, and that is provided in a limited way by mechanisms for link inhibition in some contexts of T-joints, and for link creation in cases of matching T:s (continuing, obstructed objects).

A body may sometimes be correctly identified even though an interior line be missing, provided the resulting regions get linked together strongly enough.

By the same token several objects may be clumped together into one, due to missing exterior lines.

Basing segmentation on the formation of links between regions makes this program very sensitive to imperfections.

Thus Guzman's program is dependent on a very clever preprocessor, unrealistically clever, in fact. I shall get back to that subject later in this section, but let us just list a few things such a preprocessor is supposed to be able to do. It must produce an essentially perfect line-drawing, that is, eliminate lines caused by shadow effects (as well as other spurious lines). It must also insert missing lines, and group together lines into vertices correctly, since the proper links would otherwise not be formed. No small task!

The results of Guzman's work seemed impressive. His program successfully analyzed very complex, carefully constructed scenes. One tends to forget that those scenes are not "live", and that the results are in that sense not immediately and implicitly relevant to the practical problem of computer vision.

[Winston 1970]:

Based on perfect (synthetic) line-drawings and region analysis, Guzman's program was used by Winston in a system that analyzes scenes structurally, and learns structural descriptions from examples.

Since Guzman several people have chosen to work on the same subproblem, namely the interpretation of an essentially perfect line-drawing (with or without shadow lines) as a conglomerate of objects.

[Orban 1970]:

Orban provided some shadow-eliminating heuristics that could be used in conjunction with Guzman's program, in a preprocessing stage. Those heuristics were local in nature and based on the observation that joints caused by shadows often are X:s and T:s, and that such joints are often chainwise linked together.

[Huffman 1969] & [Clowes 1971]:

These two authors devised labelling schemes to catalogue the possible interpretations of vertices that may be found in perfect, shadowless line-drawings of scenes of trihedral objects. Such labellings may serve to provide more global contexts for segmentation processes.

Such and related concepts were utilized by Falk and expanded by Waltz.

[Falk 1970]:

In the context of the Stanford Hand-Eye Project [Feldman et al 1969], Falk embarked or the development of programs to "interpret imperfect line-drawings as three-dimensional scenes".

Utilizing a vertex labelling scheme related to Huffman's ideas, Falk devised heuristics for body separation, which work for more general scenes than those of Guzman, in asmuch as some cases of missing lines, or parts of lines (at object intersections), do not cause the program to make erroneous decisions.

After body separation, such lines may subsequently be detected and inserted. The program uses the vertex labels to form links between the lines in the drawing, and the bodies are defined in terms of such links. The assignment of regions to objects, and possibly of dividing regions between objects, is a secondary problem.

Note that basing segmentation on links among lines, rather than between regions, makes this approach less error sensitive than Guzman's.

Following body segmentation, some simple heuristics are used in determining occlusion relations over the scene, and the extracted bodies (which may be partially occluded) are completed as far as possible, based on collinearities and extension-vertices.
Base edges are then found, and support relations are inferred. Such data is used in the determination of locations in space, below.

The recognition part of the system works with a programmed set of fixed size models, for which the numbers of faces and vertices are stored for each different view, along with the number of sides for each visible face. Such properties are compared with those of bodies in the scene, giving a certain list of possible matches for each body. Secondarily, the nature of the regions is used in order to reduce such lists, and the final choice is based on physical properties - lengths and angles - which are computed from the monocular view, using hypotheses of ground plane or object support. The use of objects supporting objects (flat on top) represents an extension of Roberts work, which only used the ground plane assumption. If an object cannot be recognized, a second attempt is made, using relaxed parameters.

The identities and locations in space of the recognized objects are now known, and the objects may be back-projected and compared with the original line-drawing. Techniques akin to those of Roberts are used here, including a fairly simple hidden-line eliminator. The correspondence of the original and projected drawing is evaluated, line for line, based on some parametric tolerances regarding the number and nature as well as the closeness of coincidences between original and back-projected lines.

Falk's program is related to both Roberts' (the use of models) and Guzman's (the implementation of body segmentation). The combination of those techniques is an interesting idea. The program is somewhat less error sensitive and more practically useful than Guzman's, and it has been successfully demonstrated on live data (using a preprocessor coded

by yours truly). However, due to the way segmentation is implemented - unrelated to recognition - it shares the weakness of Guzman's program (to a great extent) of being unable to cope with realistically imperfect line-drawings. Like that program, it is shadow and noise sensitive.

[Waltz 1972]:

The concepts of vertex labelling introduced by Huffman have been extended by Waltz, who not only handles, but actually also utilizes, shadows in the process of segmenting the scene. Here, too, the original data is a line-drawing, which is assumed essentially perfect. The following is a brief sketch of how the program works.

First the vertices in the scene are labelled according to all their possible interpretations, given that the scene consists of convex, trihedral objects. Each label at a vertex assigns a specific label to each one of its lines. Such line labels cover most of the possible edge interpretations in a three-dimensional scene (convex, concave, bounding, obscuring, crack, shadow), and they also cover the lighting conditions on the sides.

Waltz now applies a filtering program which checks the inter-consistency of the two sets of vertex labels for each line, deleting inconsistent labels (i.e. where the line labels could not agree). This filtering program was found to assign unique labels in a surprisingly large number of cases. If the labelling is not unique, a full tree-search for consistency is performed over the entire scene, and inconsistent labels are deleted.

The resulting labelling determines the segmentation(s) of the scene. If not unique, the labelling gives rise to several possible interpretations, which is one of the strong points of the program. It doesn't jump to conclusions.

Waltz' work contains some of the more amazing case analysis I have seen. The specificity of labellings aids in the segmentation process, but also makes the program sensitive to imperfections in the line-drawing. The program handles shadows, in fact categorizes them as such, but those have to be consistent as well as the rest of the lines. In other words we still have essentially the requirement for perfect line-drawings, though this time with (perfect) shadows.

Some facilities for dealing with missing lines are included in the program. More precisely, the case of a missing interior line of an object is treated, simply by including such special case possibilities in the labelling scheme.

Waltz' program is so far the most elaborate line-drawing segmenter in existence. It represents a radical departure from earlier (local-heuristic) schemes, in that the entire context is utilized and the scene is interpreted as a whole. That is an important achievement, in my opinion.

However, in order to be practically useful, the program would require an unrealistically clever preprocessor, a need it shares with all segmenters discussed here so far. It is dependent on very special rules

regarding the labels (shadows, background, etc.), which makes it error sensitive. Furthermore the order in which the labelling is performed may be crucial to the final result.

This concludes the discussion of systems based on the assumption of essentially perfect or almost perfect line-drawings (with or without shadows). Comparatively few people have ventured into the messy realities of live scenes - fewer have emerged with anywhere near spectacular results.

Some preprocessors:

Visual preprocessors have been constructed by Binford [Binford 1970], Brice and Fennema [Brice & Fennema 1969], Hueckel [Hueckel 1971 & 1973] (edge-finder), Pingle [Pingle & Tenenbaum 1971], Baumgart (cf. end of this subsection), and others.

Tenenbaum [Tenenbaum 1970] fathered a substantial thesis on accomodation in vision, including work on edge (line) verification and depth through variable focus.

My own experiences in the field of endeavour of preprocessing will be discussed shortly.

I shall here first briefly deal with an interesting heterarchical approach to the problem, and with a limited system using learning and recognition. I shall also mention an effort regarding vision of more general objects, and an approach using sequences of views.

[Shirai 1972]:

Shirai constructed a program that tries to find bodies in a scene, working directly on the digitized picture, and utilizing an initially extracted, perfect contour (the background is black, and the objects are white).

The process is heterarchical, and analyzes the data, looking for lines and vertices, with a general concept of "body" as a guide. It utilizes the information it has already obtained, in order to further complete an object. This is in contrast to hierarchical schemes, where successively higher abstractions are formed more or less in sequence, by a hierarchy of heuristic processes.

More specifically, the program looks for lines at concave junctions in the contour, or in other interesting places. Having found evidence of a line, it tracks along that line, looking for vertices or extensions, determining the implications of its findings as far as the concept of object goes. The global context (of object) enables parameter threshold adjustments according to current search contexts.

Shirai's program is an interesting and promising effort toward a heterarchical vision system. The idea of having recourse to the original intensity information throughout the process of segmentation is a good one, although not new, of course.

The program does not work in the presence of shadows or other

detrimental effects, and is not general enough for concave objects. However, in simple scenes, under ideal lighting conditions, it should do an adequate job. It is therefore a member in the sparsely populated class of practically applicable vision systems. So is the final related program to be described here.

[Underwood & Coates 1972]:

This work is related to mine, in that it uses learning and recognition. It is a limited vision system, working only with single objects, which are planar faced and convex.

Interactively with an image dissector camera, an edge-line-drawing is obtained. Regions are then found and their connectivity investigated.

During the learning phase, the program is presented with views showing all the different surfaces of an object, and it is able to form an internal, complete model based on topology and on certain projection invariant shape measures for the faces. Those topology models represent planar unfoldings of the objects, except for actual surface sizes.

Equipped with a set of previously learned models, the program is subsequently able to match any view of such an object with one or more of the models, using the topology and the number of sides of each surface in view. If there are several matches, the shape measures are compared in order to form probabilistic estimates of the closeness of fit.

As a vision system, it is limited to scenes of single objects, but it is complete in the sense that it goes all the way from camera input to recognition. The preprocessing phase of the program is required to produce perfect line-drawings, otherwise the recognition (or learning) will not work.
The idea of "learning by looking" is an appealing one.

[Agin 1972]:

Agin has provided some interesting initial work on the representation of curved objects, and the use of laser ranging to obtain depth information.

A curved object is represented through its typical cross-sections along a main axis. The laser is utilized in mapping out the surface curvature. As mentioned, Agin also notes the possible use of a laser to get depth information in the context of a system like the one presented in this paper. I definitely agree to that as a good idea, as indicated in the section on future work (Section 12).

[Baumgart 1974]:

Baumgart's system, which is currently under development, uses sequences of views (for instance by rotation) in analyzing scenes. Besides being able to provide 3D information, this process often tends to neutralize the effects of shadows, glare, noise, etc. The edge-finding stage of preprocessing is based on thresholding and merging.

This concludes the brief outline of related efforts in computer vision.

3.2 OWN EXPERIENCES - THE MAD QUEST

"Mine is a long and sad tale..."

As we have seen, the usual first step in interpreting a digitized TV-raster by computer (as a visual scene) has been to condense the information, to abstract relevant parts of it and thus get a smaller database, and one that is more conveniently formatted for further analysis.

The traditional way of abstracting and condensing such information is to apply a brightness-discontinuity detecting operator over the entire picture, analyzing a small fraction of it at a time. This provides a map of the relevant parts, namely where the brightness changes occur, and where we may hope to find (for instance) outlines of objects.

Roberts used this approach, followed by a line-fit, and it is used today (ten years later), at the Stanford Hand-Eye Project. Our edge-operator, however, is a much more powerful one (described in [Hueckel 1971 & 1973]).

The initial line-fit further reduces the database, further abstracts relevant information, and further renders it a format suitable for

interpretation. These things are described in not too much detail in Section 7.

We now come to a somewhat crucial point, namely:

What is the general nature of such initial line-drawings? To what extent can we expect to rely on their information? Is everything there, that should be there? More?

Some alternative answers are: "Hm ..", "Yes!", "No!", "(Mumble)..", "Maybe(?)", ...

I shall get back to those questions, but let us for a moment assume that the line-drawing is perfect. In such a case there are no problems. We group the lines into vertices, based on the (small) intersection distances, detect T-joints, find closed regions. All of that becomes more or less trivial. However, the task of interpreting the jumble of regions (or lines and vertices) as a jumble of objects is non-trivial. We have seen several different approaches to that problem, in the preceding subsection.

It was tempting, when I started working in this field a few years back, to use existing programs as building blocks for a system that would work on live scenes. The Hueckel edge-finder existed in a powerful enough incarnation. We had a copy of Guzman's program (boldly called "SEE"). I set out to investigate what kinds of results one could obtain, using the edge-drawings to produce the highest possible quality line-drawings

(with reasonable effort), and then entrusting those to "SEE" for segmentation.

Amazingly enough, the program I came up with worked reasonably well for simple scenes [Grape 1969 and 1970]. By then it had been elaborated considerably.

The following table briefly describes the flow of the final version of that preprocessor (K stands for Kluge):

K1. Edge detection.

K2. Initial line-fit.

K3. Formation of initial vertices, based on closeness of edges.

K4. Formation of exhaustive cross reference tables, for each line-end listing the best 3 extension intersections, blocking lines, possible cuts, nearest collinear line.

K5. Using that cross reference table to form secondary vertices (iteratively, in a parameter relaxing loop), using brightness information as an additional criterion.

K6. Grouping of initial, secondary, and final vertices into final vertices (i.e. iteratively), using different

heuristics according to appearances of the constituent vertices. This necessitated fairly elaborate heuristics to prevent the line-drawing from self-destructing, since moving a line (to accomodate a new vertex) might cause secondary movements to existing vertices, especially in connection with T-joints. It was solved essentially by allowing the line-drawing to float, only describing the connectivity, until all vertices were determined, at which time their constituent lines were all weighted to provide the best possible vertex coordinates.

K7. Finding connected paths, outsides and insides.

K8. Determining closed regions.

K9. Line prediction and verification. This loop was based on criteria for well-shaped-ness of regions, and on parallelogram completion. Predicted lines were accepted or rejected on the basis of the number of edge points found inside an elliptic (or sometimes rhombic or rectangular) operator of parametric width, and with the predicted line as main axis.

K10. Producing the resulting line-drawing in the format required by "SEE" (also known as "Guzmanizing").

The final building block in that vision system (at that time) was then provided by "SEE".

Note that step K9 constituted a step away from sequentialism, and toward modelism. The model, in this case, was the concept of a well-shaped region. Note also the interaction with the original edge-data.

In the meantime Falk wrote his program (described above), for which my preprocessor was expected to provide reliable input. The prediction - verification loop was then unfortunately not yet accessible for common use, and consequently Falk had more trouble than necessary in obtaining scenes on which to demonstrate his program. Luckily, my system incorporated facilities for editing line-drawings.

It became increasingly clear to me that the perfect line-drawing could not be achieved by a preprocessor based on local heuristics. Most people in vision work probably agree, by now. I never expected to be able to provide such line-drawings in general (by a long shot), but the messiness of live data exceeded my expectations. Due to disturbances in the scene, as well as hardware glitches, one is almost always faced with defective initial data (for scenes of reasonable complexity) in such a way as to make locally based decisions impossible. Experience, if nothing else, has demonstrated the need for global knowledge of some form, even at the intermediate and low levels of computer vision.

The next subsection is an attempt to analyze that need for global knowledge. Possible solutions are discussed, particularly as provided by the deployment of prototype-driven schemes.

3.3 SEQUENTIALISM VERSUS MODELISM

The following is a clarification of the title of this subsection.

The term sequentialism refers to the common method of sequentially finding successively higher abstractions (starting with the digitized image, and possibly ending with body-segmentation), where an abstraction phase cannot be repeated at the request of some higher-level procedure, using global contexts. Decisions in sequentialism are often of necessity based on local contexts.

By modelism I here refer to the utilization of global knowledge at various levels (by the use of a concrete set of models, or an abstract, driving concept), in such a way that low-level decisions may be subject to revision, based on the findings of higher-level processes, or that such knowledge is used to drive those stages in the vision process.

We have noted the assumption of essentially perfect line-drawings for several vision projects described previously. Those provide examples of sequentialism. On the other hand, for instance, Shirai's program works somewhat in the spirit of modelism, in asmuch as it interprets the picture with the concept (model) of object as a driver of the process, and actively looks for objects in the scene from the beginning.

The idea of actively looking for things, based on various clues present in a tentative initial line-drawing of the scene, is the basic principle behind the vision system described in this paper.

Let me now give an example to illustrate the difficulties of vision through sequentialism.

Figure 3.1 demonstrates the hazards of locally based decisions, in the formation of vertices or in otherwise interpreting scenes.

Now take a look at Figure 3.2, which shows the complete initial line-drawing, from which the close-ups in the previous figure are excerpts.

Being human, we understand this scene very quickly, now that we can see all of it. But that is exactly what it takes here! Not necessarily to be human, but the ability to see global relationships, and to be able to interpret those, even in the presence of spurious data and the absence of lines that should have been there.

The principles of human vision are not necessarily something we want to imitate, in order to create a computer vision system. We simply couldn't! But that great Master, Evolution, has had a long time at his disposal, and we should learn as much as possible from our own ways of visually perceiving the world.

I think we tend to see the whole before the details, as a rule. The examples I have just given certainly support such a theory. Seeing the global relationships, we are able to correctly interpret or classify the elements in the partial pictures. Sequentialism, of course, attempts to do exactly the opposite, namely classify the local relationships, and from them somehow to infer the whole.

Figure 3.1

Difficulties in sequentialism

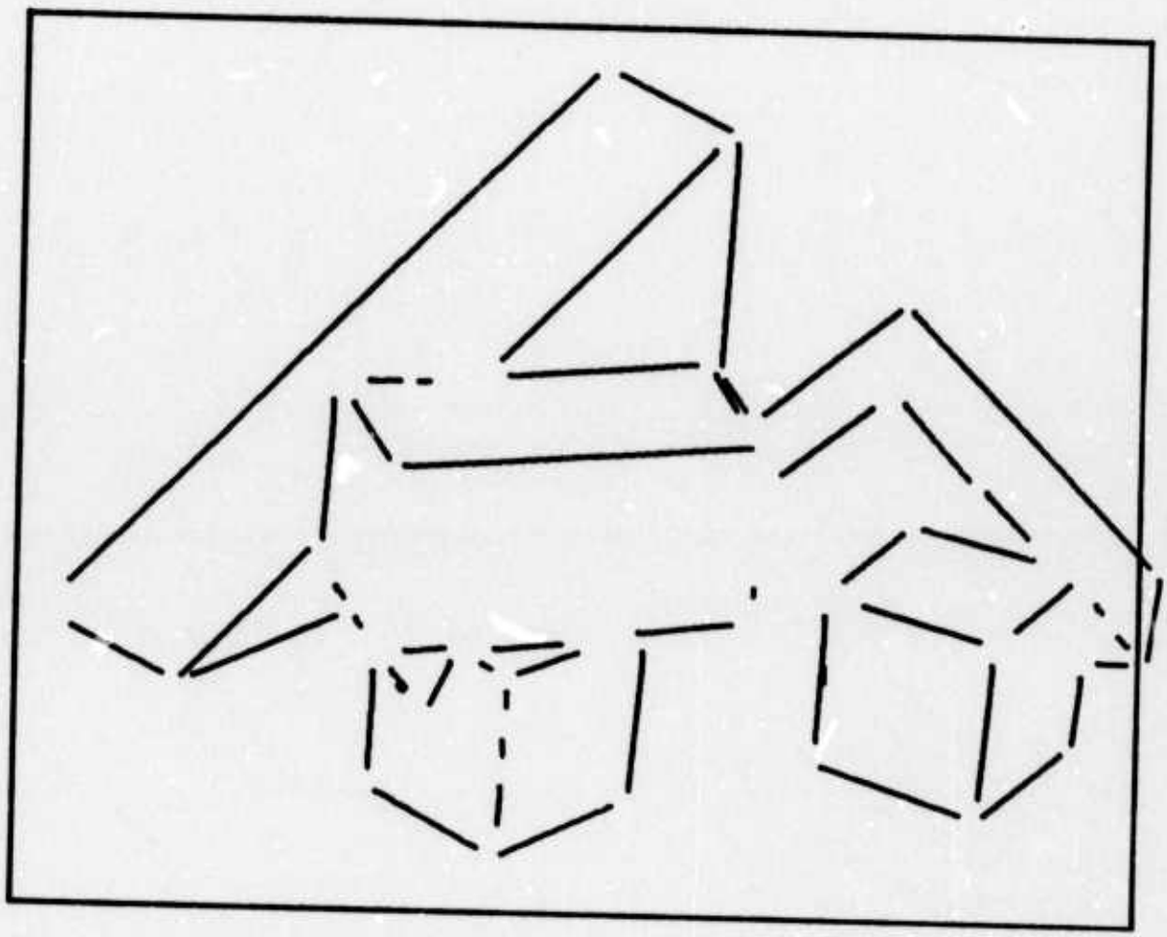


Figure 3.2

Advantages of modelism

In my opinion, and judging from my own experiences, sequentialism is doomed to failure, at least in dealing with realistically complex visual scenes. The concept of global knowledge is crucial, not only at low and intermediate levels (such as vertex formation) but also at the level of three-dimensional interpretation. Here, global concepts enter in the form of support theory, understanding of depth and occlusion relations, etc. Ideally, I think, this should also interact down to the lower levels.

I am now about to embark on the main purpose of this paper, the presentation of a vision system in the modelistic spirit. It learns its prototypes, and the understanding of the scenes is based on recognition. It tends to see global structures in somewhat the same way I do, and is therefore relatively insensitive to imperfections in the scene representations.

4.0 STRATEGY OVERVIEW

4.1 GENERALITIES

It should be clear by now that the purpose of the present intermediate-level vision system is not to produce a "perfect line-drawing", to be further processed by "higher-level" programs. The perfect(ed) line-drawing rather has the character of an optional by-product, which is nice to display as a demonstration perhaps, but which is not necessary for the purpose of the computer "understanding" and/or being able to manipulate the scene.

The purpose of the system presented here is to parse the scene, "parsing" being defined as determining the nature and location of the partaking objects as expressed in their two-dimensional projections. It leaves the aspects of three-dimensional positions and relations to a higher-level program, which is as yet non-existent as a whole, but for which parts of Falk's work may be adapted (cf. Subsection 3.1). The development of such a system is currently under way at the Stanford Hand-Eye Project.

I shall only briefly discuss the role and reason for the dichotomy into different levels of models, the main discussion having been presented in Subsection 3.3. The ILV-system proposed in this paper uses 2D prototypes, which are perspectively consistent projections of the

Preceding page blank

different views of their parent objects. This method has the advantages of simplicity in representation, ease in feature-extraction and convenience in mapping. Basically this system is an experimental subset of a possible, more extensive system based on 3D models. From such models the 2D prototypes might easily be extracted through systematic projections. Some aspects of an extended system are treated in Section 12. A detailed account of the 2D models is to be found in Section 6.

The present system runs a complete parse on the entire scene, stopping only when the scene is exhausted. One good reason for this behaviour is that there is nothing else for it to do, since the higher level (3D) package does not exist. Given the presence of such a higher level program, it may prove desirable to drive the parse from that extended system, with full utilization of the concept of 3D and with the necessary support theorems, etc., checking each mapping individually.

The concept of generality has been of considerable importance to the author during the course of this undertaking, and the present 2D system does (within its scope) have that desirable property. Input, analysis, and learning of prototypes is fully automatic. There is no special case analysis (with the exception of perspectively degenerate views) other than that which is implicit in the structuring of the feature hierarchy (Section 6), which influences the prediction - verification elements of the matching program (Section 9). Perspectively degenerate views (Subsection 6.8) have required some degree of special treatment.

Since it is out of the question to exhaust all mapping possibilities

between the scene and the prototypes on a random basis, the utilization of easily extracted, easily mapped and recognized features becomes imperative. The features serve as keys for the matching process, and are also used throughout that process for purposes of prediction and verification.

The overall structure of the system is built up in the following way:

1. Preprocessing (Section 7).

2. Parsing (Section 8).

(3.) Object completion (Section 10).

The third of those phases is an unimplemented possibility. Each of these processes displays some non-sequential behaviour, notably the parser, which uses a non-sequential (recursive) matching program. The term "non-sequential" is used here to stress that the process is different from that of "sequential abstractions".

Many examples of the proceedings of this system are provided in Section 11, and I strongly suggest that you take a preliminary look at those now, before continuing in the text (unless you are an expert, and know what is going on). This should make the rest of the presentation easier to follow.

4.2 STRATEGIES

More precisely the strategy is as follows:

1. Fit lines to initial edge-data, iteratively and conservatively.

2. Parse the resulting line-drawing, in a looping process, each time finding the best possible match between elements of the scene and some prototype, isolating that mapping, modifying the scene (by removing the lines), and iterating. This process ends when there are no more possible mappings.

(3.) Bring isolated, incomplete mappings (parts of prototypes) back into the scene, one by one, in order of decreasing complexity. Then investigate for possible extensions of the mappings (taken one by one). This program could use the same principles as (and indeed parts of) the mapping program.

The thus obtained interpretation of the line-drawing, as a 2D projection of a 3D scene, consists of a set of disjoint, possibly only partially mapped, prototype instances.

At this stage it would be possible to further investigate the union of these instances, comparing with the initial data, thus determining most

of the occlusion relationships between the different objects. We would then quite often be able to obtain a corrected (I refrain from saying "perfect") line-drawing. The reason I have abstained from implementing such heuristics is simply that the program would not be dependable enough, it would be somewhat klugy in nature, and it would be uncalled for in the context of an extended system, where such relationships (as mentioned earlier) would be much more elegantly and soundly determined on the basis of positions in space, support relations, etc.

For the present the parsing program is non-recursive, i.e. it accepts the currently best match between scene and prototypes, amends the scene accordingly, and then carries on in the same style with the amended scene. Another possibility might be to introduce recursion at that level as well (as the matching level), thus keeping a number of different alternative parses around, among which a most likely candidate may be chosen (verified). The combinatorics, however, would seem rather forbidding, as would storage requirements. Furthermore there is no sufficiently established need (yet) to warrant an effort in that direction.

From this strategy overview we now turn to more detailed accounts of the component parts of the system.

5.0 FEATURES

5.1 INTRODUCTORY EXAMPLES

The use of features to provide mapping clues (and matching guides) from scene-elements to prototype elements is essential to the system presented here. I shall consequently deal with these concepts in some detail.

The general idea behind this system is one of recognizing elements encountered before, as parts of familiar things (objects). After such "first impressions" the system proceeds to verify or refute its initial theory regarding the identity of the object. The instruments of "first impressions" are called "features".

The feature hierarchy is based on the (personal) observation that we get strong visual clues (in a 2D image) from the way in which side regions (face projections) of objects come together (information that we use with the shape of the regions in order to make sense of objects).

Therefore the features have been constructed to contain extensive information about region junctions, i.e. the lines (including end-vertex constellations) of two-dimensional projections (object or prototype). While the features do not contain full shape information, they provide enough to serve as strong clues and as guides during the matching process.

Preceding page blank

In order to avoid thousands of words I hasten to give an illustrative example. Figure 5.1 shows a scene and some of the features we may find in it (heavier lines).

I shall also give an example of a prototype and the features it contains. That presentation will be followed by more precise definitions. Figure 5.2 shows a 2D projection of a parallelepiped. We see that the junction of any two faces (as given in the projection by their common line) and the corner junctions at the ends of their common edge (as given by the end-vertices of the line) presents one of only three rotationally distinct line and end-vertex constellations (in the plane), namely as given by L1, L2, and L3.

Those line constellations are examples of the basic feature, called "line-feature" (abbreviation: LF), and there are three instances of each one of them in the figure. It will become clear later why L1 and L2 are essentially different. The LF:s are directional (this will be clarified shortly), as indicated by the arrows.

There is one more level in the feature hierarchy, namely the "compound" (composite/complex/combined) feature (abbreviation: CF), which is simply an aggregate description of two connected LF:s, each of which is a ray of the other. Figure 5.3 demonstrates this.

That description shows how they are connected, and also gives additional joint information about opposing rays extending from the extreme ends. The CF is a strong discriminator, which may be seen in the prototype

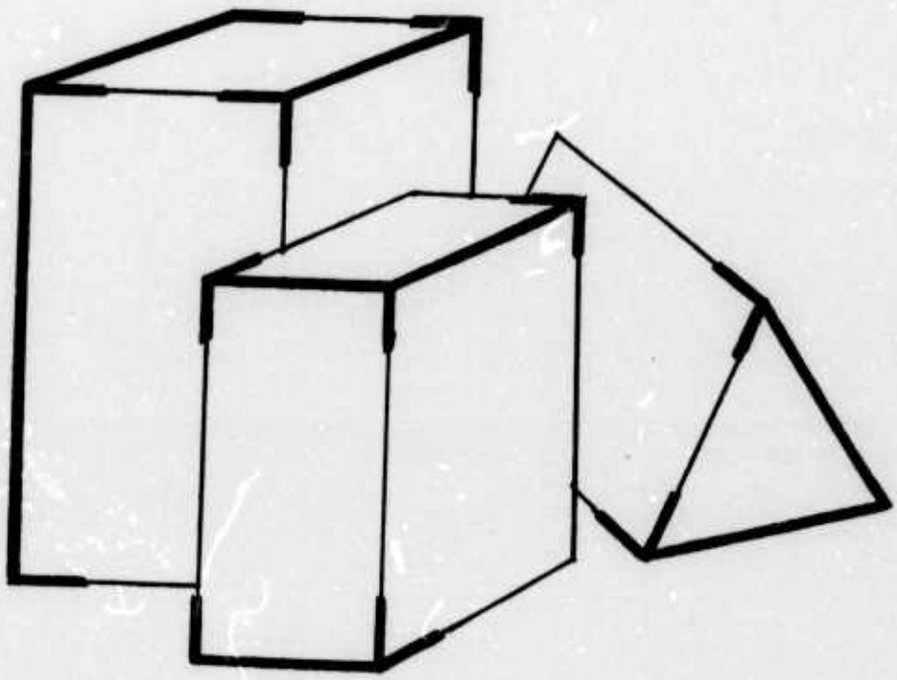


Figure 5.1

Examples of features in a scene

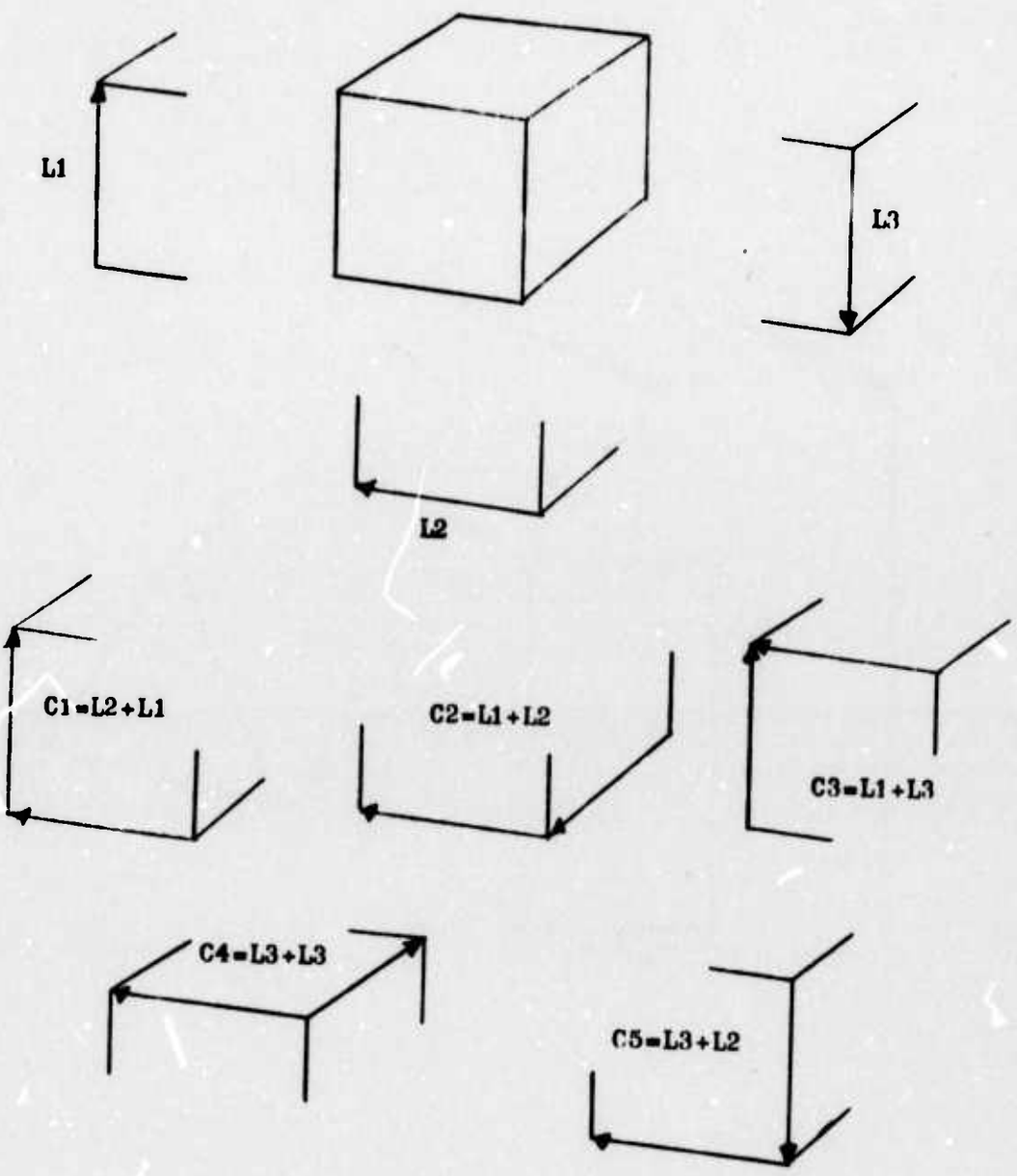


Figure 5.2

Prototype PAREP and its features

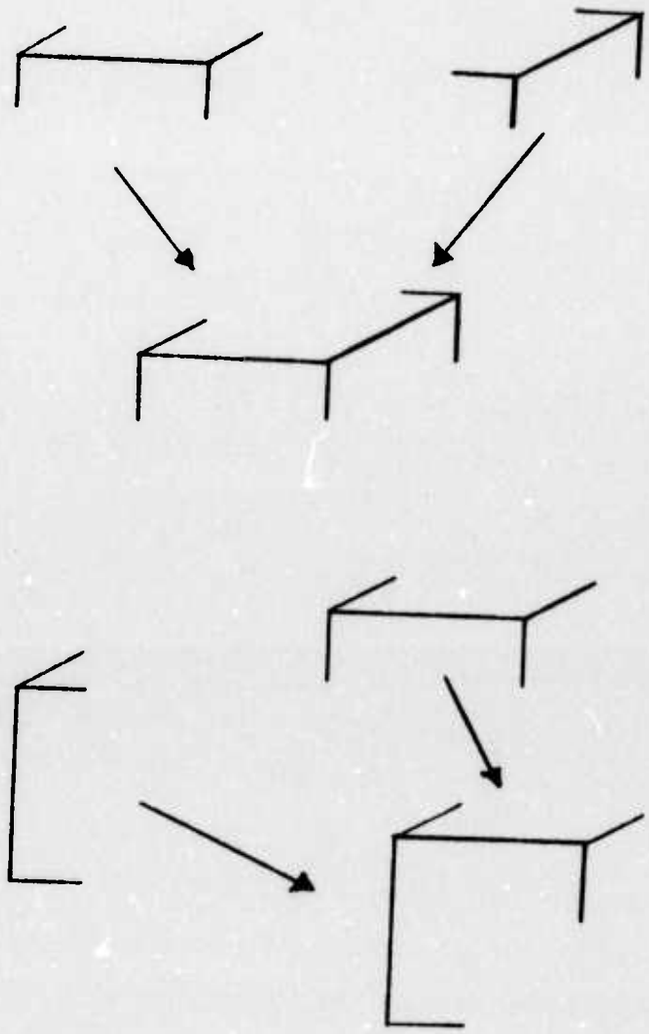


Figure 5.3

Line-features make up compound features

example (Figure 5.2) from the fact that there is one CF (C4 in the figure) that contains references to all but two of the lines in the projection of the parallelepiped. The CF:s are intended for use primarily as initializations (keys) in the matching (mapping) process.

Figure 5.2 shows all of the five different compound features of the PAREP prototype. C3 and C5 are essentially different for reasons given below.

Hoping that these examples have provided some of the flavour of the feature concept, we now proceed to more formal definitions.

5.2 FEATURE DEFINITIONS

The features conform to the following definitions:

LF0. A line-feature (LF) is an encoded description of certain basic, projectively invariant characteristics (in 2D) of the combined junctions of the side-regions of two simply connected vertices (representing corners in 3D), i.e. of a line and its end-vertices.

CF0. A compound feature (CF) is an encoded description of the same properties for three chain-wise simply connected vertices, in terms of the LF:s of the two connected lines, and additional information.

The concept of projective invariance is explained in Subsection 5.6. "Line-feature" is sometimes, and somewhat loosely, used interchangeably with "line" when the meaning is clear.

Note that in CF0 the vertices may be connected in a triangle, in which case we generally get three basically different CF:s. We may also get only one, but never two in the case of a trihedral, interestingly enough. A proof of this is given later (in Subsection 5.5), not because the result is of any use but because the case is of value as an illustration of the concepts presented here.

We proceed now to a description of the encoded information (illustrated by examples), which will be followed by discussions of some of the properties of the LF and CF. Both kinds of feature are coded into 36 bits (one word) of storage, and are largely handled by the same routines.

The line-feature consists of the following items for each direction of the line (18 bits):

LF1. LF - CF discriminator (flag).

LF2. Number of rays forming an angle >180 (measured ccw. from the ray) with this end of the parent line.

LF3. Any of those rays approximately =180?

LF4. Number of rays forming an angle of ≤180.

LF5. Any of those rays approximately =180?

LF6. Outside angle (< ≤ ≥ > 180?), measured from the last ray in item LF2 to the first ray in LF4, either of which may be the base line itself. This item shows the convexity of the vertex.

LF7. Constellation of the two opposing rays on the right-hand side of the parent line, traversed from the present end to the other. Viz., are they converging to this side, or diverging? Could they be parallel (allowing for perspective)?

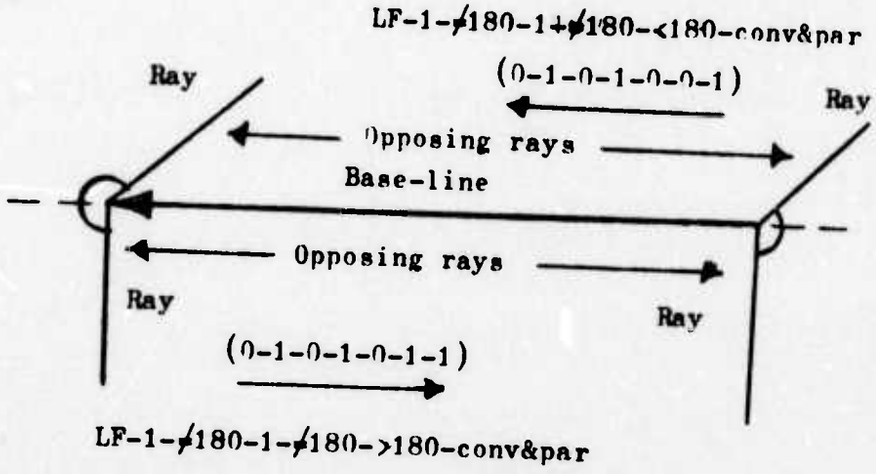
Item LF7 is of special importance for the prediction aspects of the mapping program, as the sequel will show. Figure 5.4 provides examples of LF:s and their encodings.

The compound feature contains, for each direction of traversal of the line-pair (18 bits):

CF1. CF - LF discriminator (flag).

CF2. LF identifier for first line-feature in this direction (refers to a central list of encountered LF:s).

LF-items separated by "_"



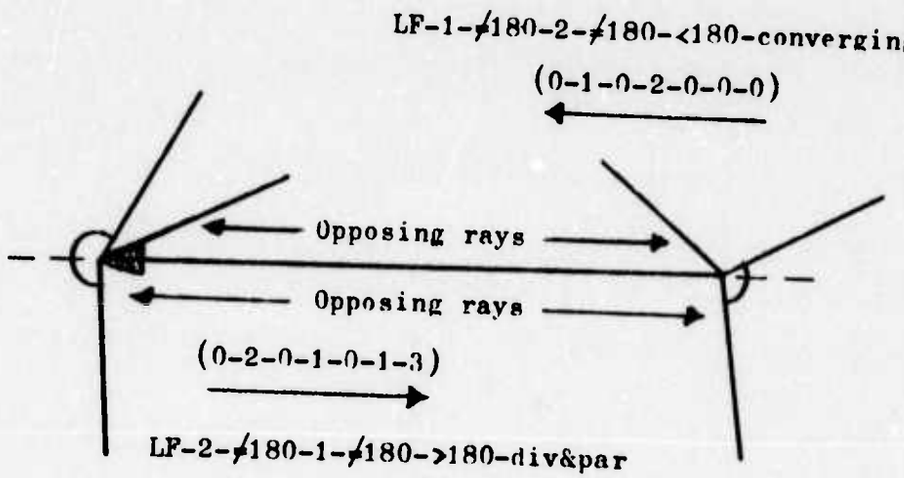


Figure 5.4

The line-feature and its encoding

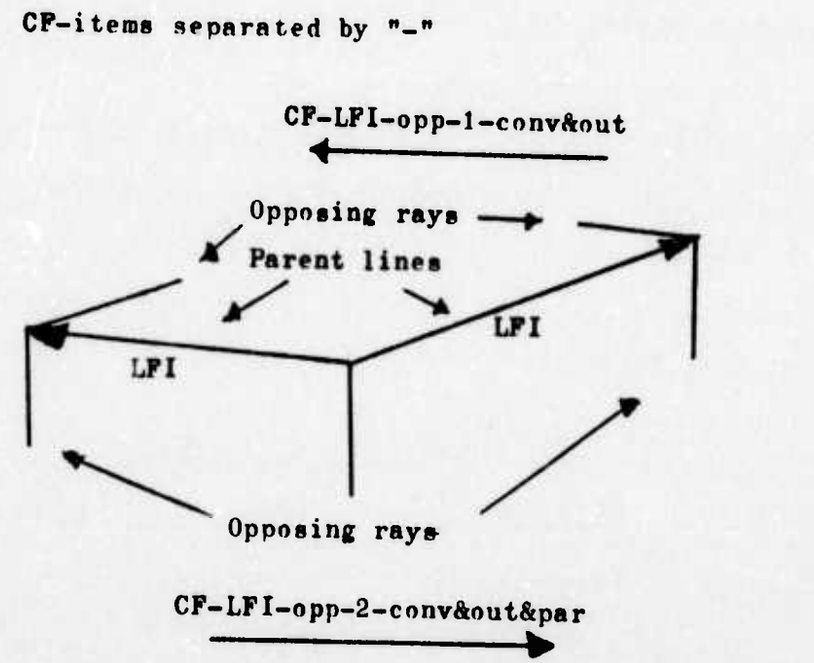
CF3. Direction in which that LF is "traversed", going toward the junction of the pair.

CF4. Position of the other parent line, ccw. around center vertex, relative to this parent line. I.e. (1 + number-of-rays-in-between).

CF5. Constellation of opposing end-rays, similarly to the corresponding LF-item, but with additional bits for collinearity, and for the direction in which these rays (would) intersect (out from - or toward - the CF).

Examples of CF:s and their encodings are provided in Figure 5.5.

Both kinds of feature are subject to an internal ordering, so that if the two halfwords (each describing one direction of traversal) are not similar (the intuitive meaning is close to the formal definition), they are ordered with the least halfword first. Similarity will be defined immediately following this. Most LF:s are ordered (directional), and we shall see that all CF:s are directional. Subsection 5.4 treats such matters in detail.



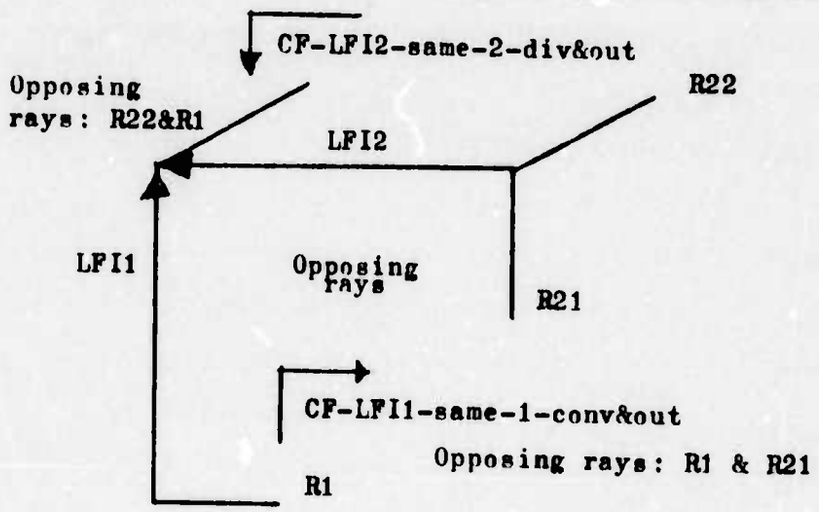


Figure 5.5

The compound feature and its encoding

5.3 THE FEATURE SIMILARITY RELATION

The basic idea behind the similarity concept is that we want to be sure that two similar features are projectively equivalent, in terms of the junctions of their affected side-regions. The important things here are the number of side-regions, their constellations at parent-line end-vertices, and also the angular convexities.

Information regarding shape of regions is of secondary importance and we allow some laxity here, as indicated by the definitional exceptions for parallelities and collinearities below. This is also inherent in the feature implementation, since there are no references to secondary ray constellations. The matching program is of course much more rigorous in such matters (Section 9).

Definitions of feature similarity:

LFS. Two line-features are said to be similar (loosely "equal" or "the same") if and only if all the items in the LF-definitions are identical, with the exceptions that "=180"-items are ignored, and convergence-divergence indicators are ignored in cases of parallelism (for opposing-ray items).

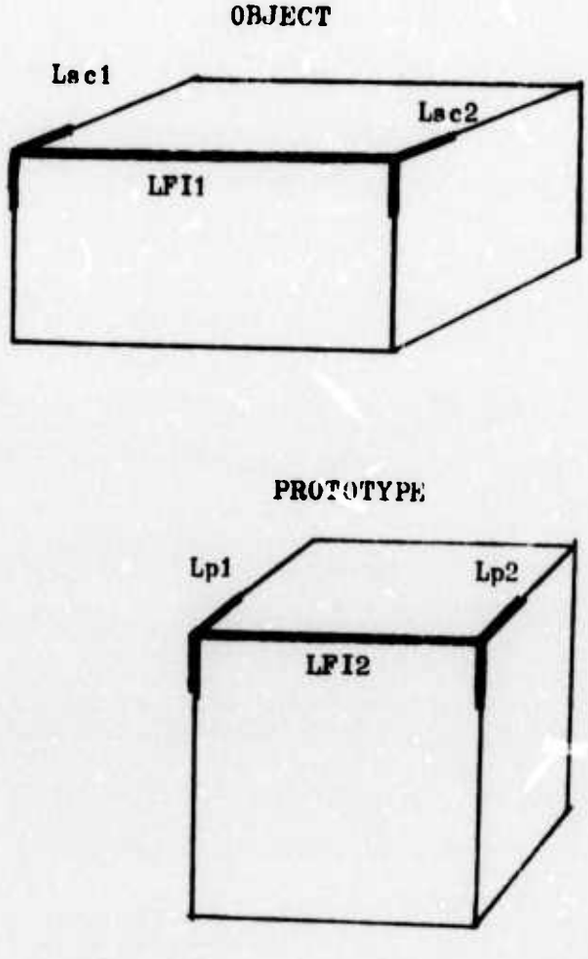
CFS. Two compound features are said to be similar if and only if all the items in the CF-definitions are identical, except that convergence-divergence indicators (for opposing rays) are ignored in cases of parallelism or collinearity.

Figure 5.6 motivates the comparison exceptions in parallelism cases. "=180"-bits are ignored here, simply because they are not reliable.

Practically, i.e. in the program, comparisons are performed through appropriate masks, using logical operations and shifts. Thus the feature handling is very efficient, and is somewhat in the nature of "hardware". From similarity tests we get information about relative magnitude of tested features, in the case where they are not similar. This is used as a basis for the internal ordering of features, as well as for the ordering of central feature reference storage.

Figure 5.7 illustrates the situation of similar features but different shapes for non-trihedral objects. The features in that figure are all similar.

In the case of trihedrals we preserve shape relations to a greater extent, see Figure 5.8. The cases where we do not have full convergence information for combinations of rays is where we have 3 or 4 rays extending from the same side of the parent line.



We want the object to match the prototype.
Thus LFI1 and LFI2 should be judged similar.
Lsc1 and Lsc2 are diverging, but approximately parallel.
Lp1 and Lp2 are converging, and approximately parallel.

Figure 5.6

Feature similarities

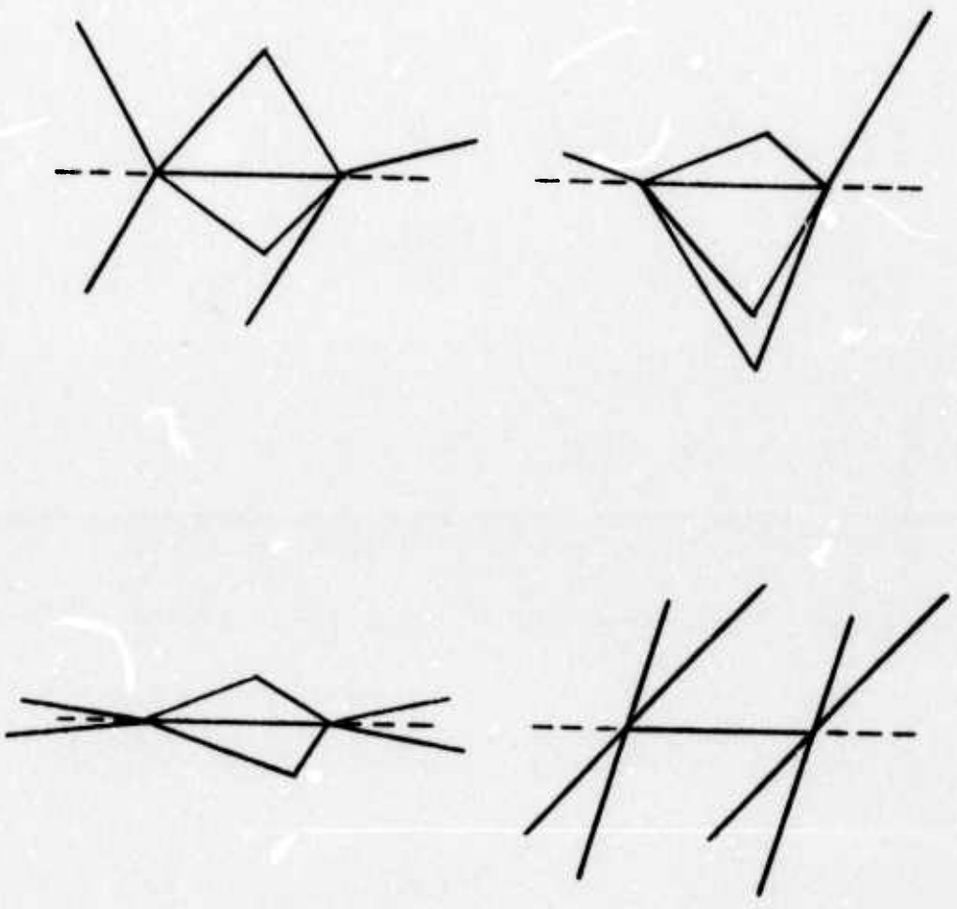


Figure 5.7

Similar non-trihedral LF:s

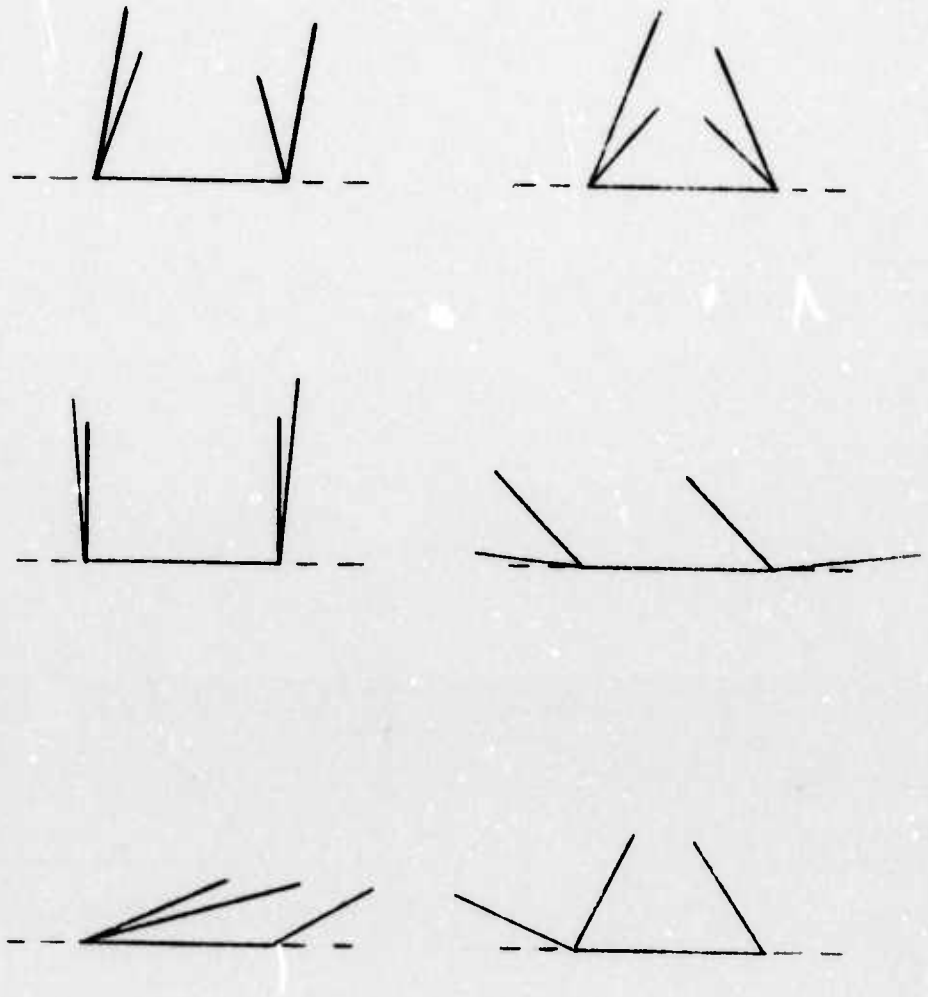


Figure 5.8

Pairs of similar trihedral LF:s

Let me point out, once more, that we do not rely on the features for (partial) shape information about regions.

In the context of matching (Section 9) we will come back to similarity, more precisely to the concept of "partially similar line-features", which is used for prediction purposes within that process.

5.4 NON-DIRECTIONAL FEATURES

Almost all commonly encountered (non-degenerate) features are directional, i.e. the half-words are not similar. Another term for this is "ordered". We shall see now what similarity of the halfwords would imply about the line-constellations. Intuitively we would expect symmetries, and that is basically what we get. At least for the LF. We shall show that there is no such thing as an unordered CF.

In the case of the non-directional LF the two halves are essentially the same, traversed in opposite directions, so that, topologically at least, we get a rotational symmetry around the center of the parent line. Figure 5.9, part (a), demonstrates - through stepwise build-up - the fact that in order to be non-directional, a convex trihedral line-feature must have exactly two rays at each vertex and on each side of the parent line, with ray convergence and outside angles in agreement. The figure also indicates some of the reasons for the various steps.

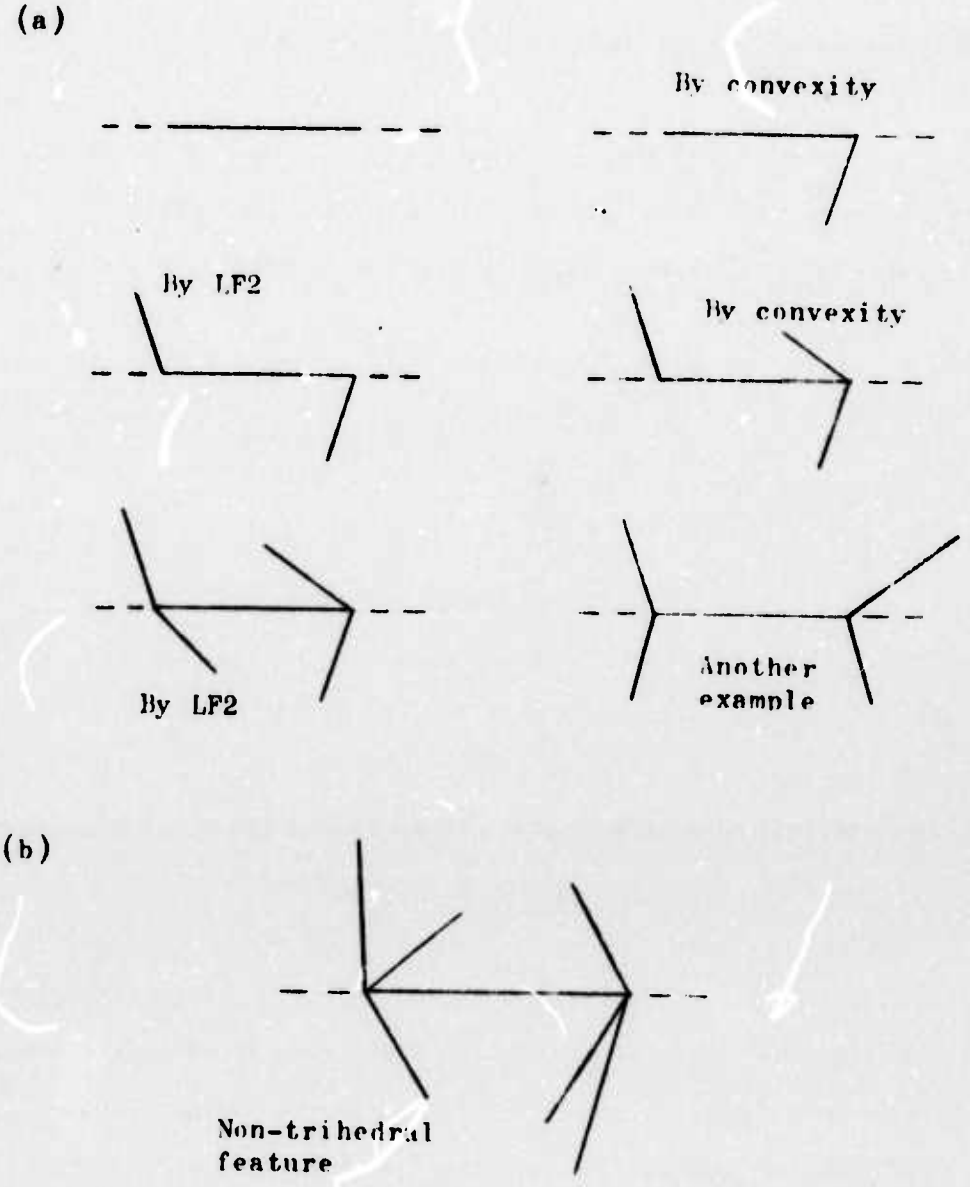


Figure 5.9

Non-directional LF:s

The same figure, part (b), shows a non-trihedral example of the general topologically rotationally symmetric case.

We now prove the following interesting property for compound features:

Theorem 1:

All compound features are directional (internally ordered).

This theorem will be a direct consequence of the following assertion:

NDCF. If a compound feature is to be non-directional, the two parent lines must be collinear.

Proof of assertion NDCF:

Let us assume that the parent lines are not collinear (see Figure 5.10). We then have:

(1) The LF:s must be similar (this is obvious from CF2).

(2) The LF:s must be non-directional.

Figure 5.10 (a) shows the parent line-pair. Let us assume, in contradiction to (2), that the LF:s are directional. Their internal ordering relative to the center vertex must then be the same (from CF-def. item CF3). This is indicated by the arrows in (b) of the fig. The

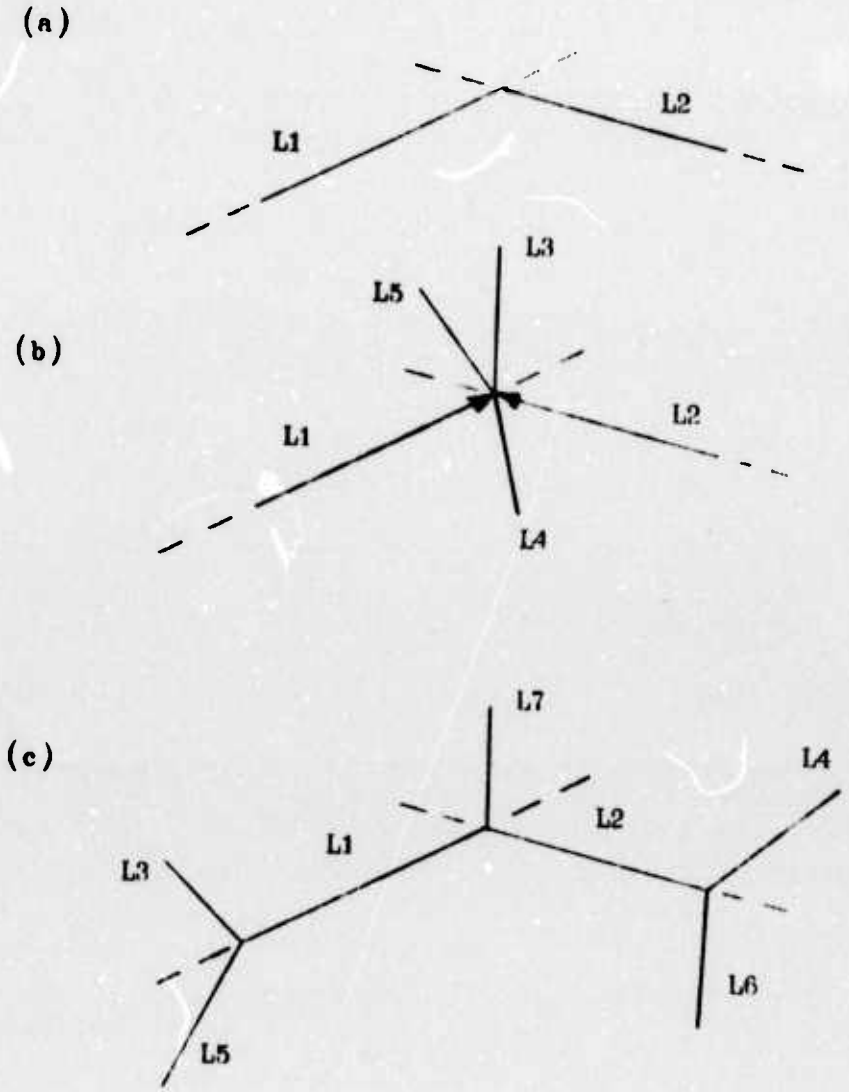


Figure 5.10

A non-collinear CF is directional

lines L3, L4, L5, ... are needed one by one according to the following argument.

The need for the existence of line L3 arises from the fact that the center vertex represents the same constellation for both LF:s. However, now we must have line L4, based on rays-in-between-parent-lines (CF4). After this we need L5, by the previous argument. Etc, etc. Clearly this process never ends, whereby we infer a contradiction.
Thus (2) has been proven.

Part (c) in the same figure shows the case where the LF:s are unordered. Lines L3 and L4 are necessary because constellations for L1 and L2 (resp.) must be rotationally symmetric. Next we find that the LF:s are no longer similar, which we try to remedy by inserting L5, L6, and L7. From then on the argument is brought back to case (b) and the center vertex.
We arrive at a contradiction, which proves NDCF.

All we need to do now to prove the theorem is to remember that the LF-definition groups collinear rays on the "≤180"-side (LF4), and that we are not making a special case of the collinearity.

Lining up two copies of the final feature in Figure 5.9 (a) gives an idea of what an unordered CF might have looked like, had it existed. The subject of degenerate views is treated in the prototype context, Subsection 6.8.

5.5 SOME RESULTING FEATURE IDIOSYNCRASIES

It may be seen now, that the LF-encodings for the two constellations L1 and L2 in Figure 5.2 are completely different, being ordered in opposite directions to start with, as Figure 5.11 demonstrates. This is very well - they should be - since the two line-constellations are essentially different. There is no way in which one of them can be made to cover the other by a rotation-translation (in two dimensions), a basic inherent principle of the matching process.

Taking the line-constellations by themselves, as conglomerates of lines in space, we may achieve a match by also rotating one of them in a plane at an angle to that of the page. This would correspond to looking at the back of the object. In the general case, of course, we know nothing about the back of an object, and can make no assumptions regarding its features. If the back differs, a different 2D prototype is created for that view. The same line of reasoning applies to the CF:s, C1 and C2.

We have noted that CF:s are directional, and that the number of rays from one parent line to the next, ccw. around the center vertex, is one of the distinguishing items of information. However, the convexity of the angle between the parent lines is not (for any direction of traversal), and we now proceed to show the reason for omitting it.

Theorem 2:

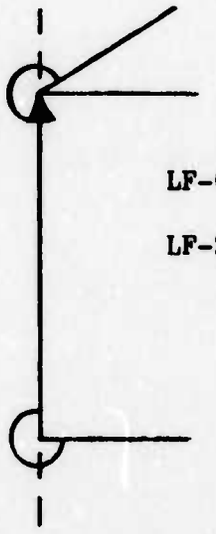
Two similar (Subsection 5.3) compound features have the same angular convexity of parent lines at the center vertex in the directions of traversal.

Proof:

Assume that all partaking features are directional. Theorem 1 (Subsection 5.4) showed that the CF:s are. The case where the LF:s are not is handled analoguously to that case in the proof of Theorem 1.

Figure 5.12 illustrates the steps below, with parts I and II showing the actions in parallel. Assume that the CF:s consist of the LF:s LFI1 and LFI2 (those may or may not be similar). Since LFI1 is directional, and we know that the LF direction bits (CF3) in the CF:s must be equal, the two instances of LFI1 are both pointed the same way with reference to the center vertices, as indicated by the arrows in the figure. But then items LF2 and LF4 in the line-feature definition (Subsection 5.2) necessitate the presence of the lines L1 and L2, respectively, as shown in the second step, (b) in the figure.

Now item CF4 (orbital distance) needs line L3 in order for CFI1 and CFI2 to be similar. Then LF2 and LF4 demand L4, etc. We reach a state of contradiction, since we can never satisfy the CF definition and the LF definition simultaneously.



LF-0-≠180-1-≠180->180-...&par

LF-2-≠180-0-≠180->180-div

LF-items separated by "-"

LF-0-≠180-2-≠180->180-...&par

LF-1-≠180-0-≠180->180-div

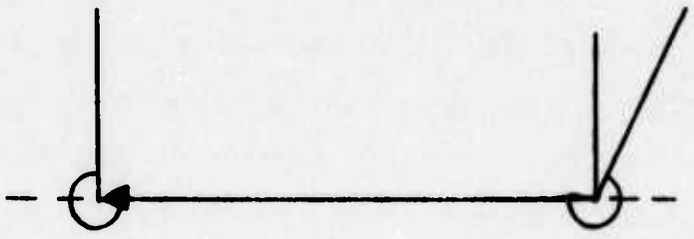
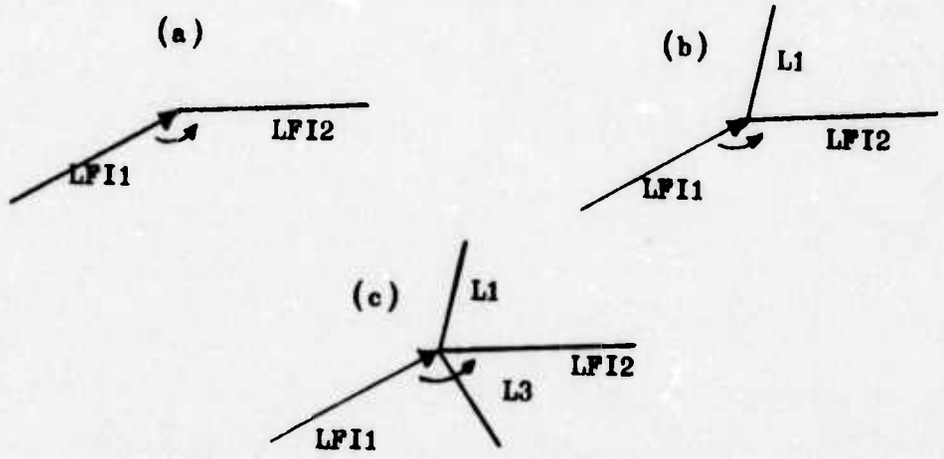


Figure 5.11

Two non-similar LF:s

I



II

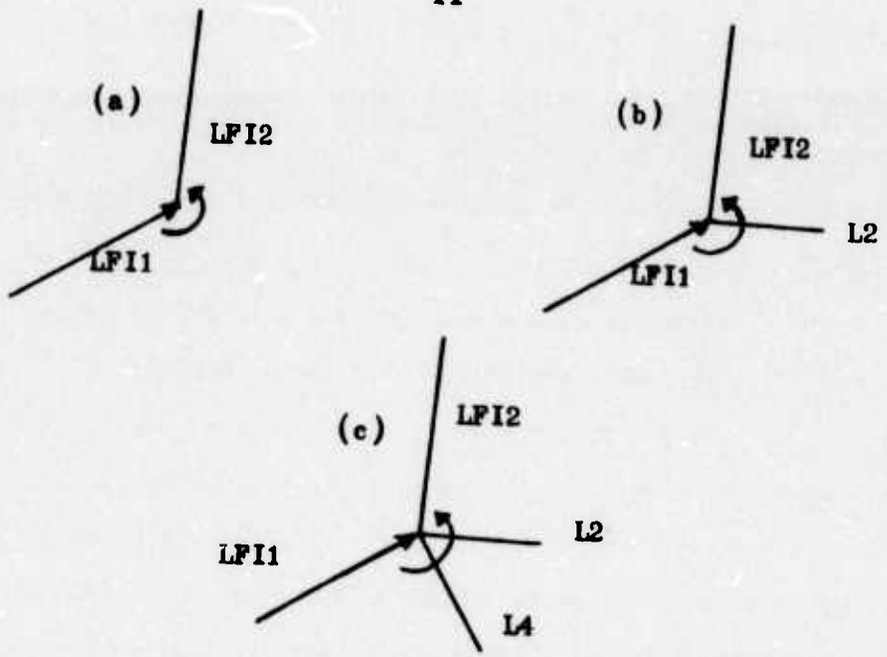


Figure 5.12

CF:s and angular convexity at center

This concludes the proof.

As a further illustration of feature idiosyncrasies we now prove that, in the convex trihedral case, for three lines forming a triangle (within a structure of other lines), we get either three similar CF:s or three mutually different ones, never two similar, with the third non-similar to those.

Proof:

(Refer to Figure 5.13). Part (a) illustrates the case of one LF and one CF only. Now (part (b), rays omitted) assume that the CF:s L1&L3 and L2&L3 are similar (other cases are treated analoguously), with two different LF:s present. Then, by Theorem 2, we get a contradiction on angular convexity.

It follows that the LF:s must all be the same, and that the CF:s are traversed L2&L3, L3&L1, as shown in part (c) of the figure. In order to have L1&L2 non-similar to the others, we must add at least one line somewhere, say L4. Part (d) exemplifies this, for a direction of LFI1 of L1. But if the line-features are all similar, we must insert L5 and L6 (and then more), as shown in (e). However, this would be impossible in the case of a convex trihedral, unless the extra lines are positioned symmetrically. But in that case the CF:s would all be similar again (part (f)). The case where LFI1 is non-directional is treated analoguously.

This concludes the proof.

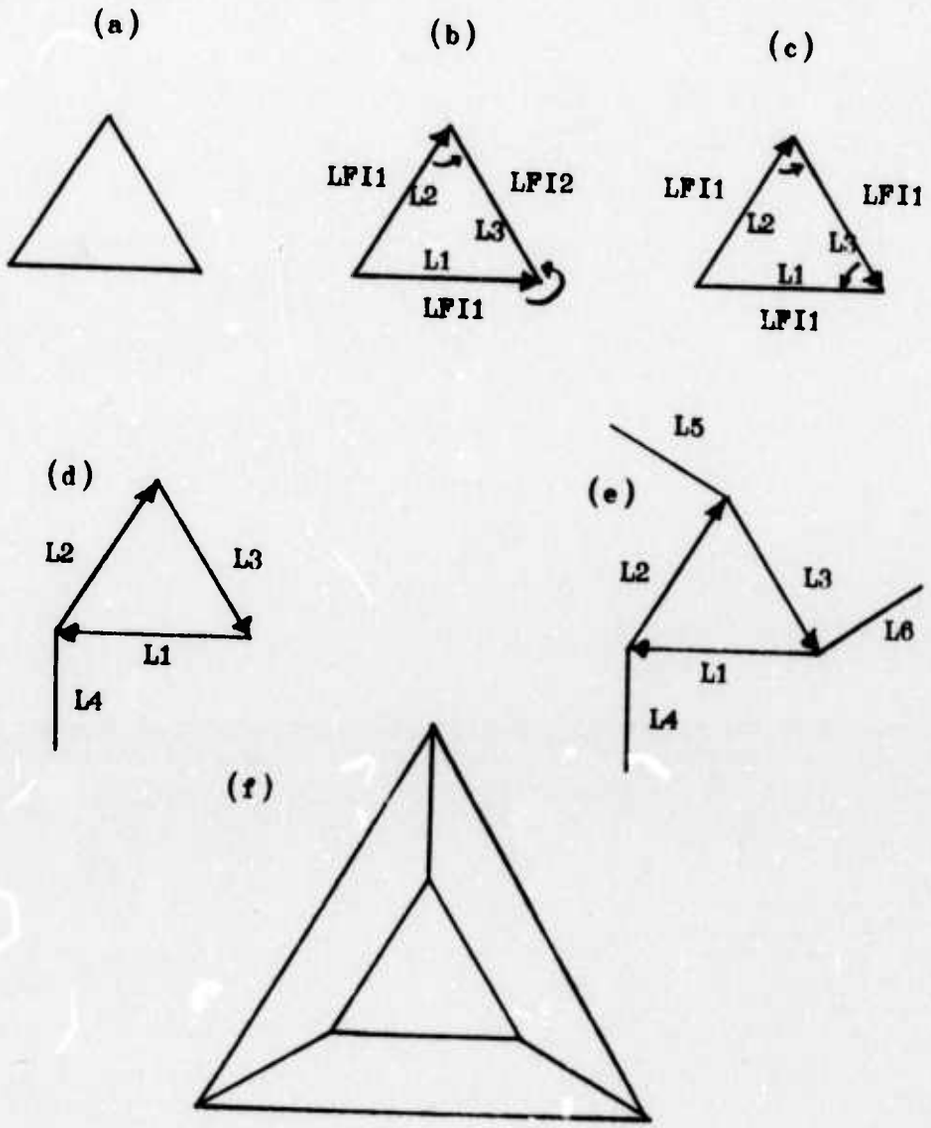


Figure 5.13

Triangularly connected CF:s

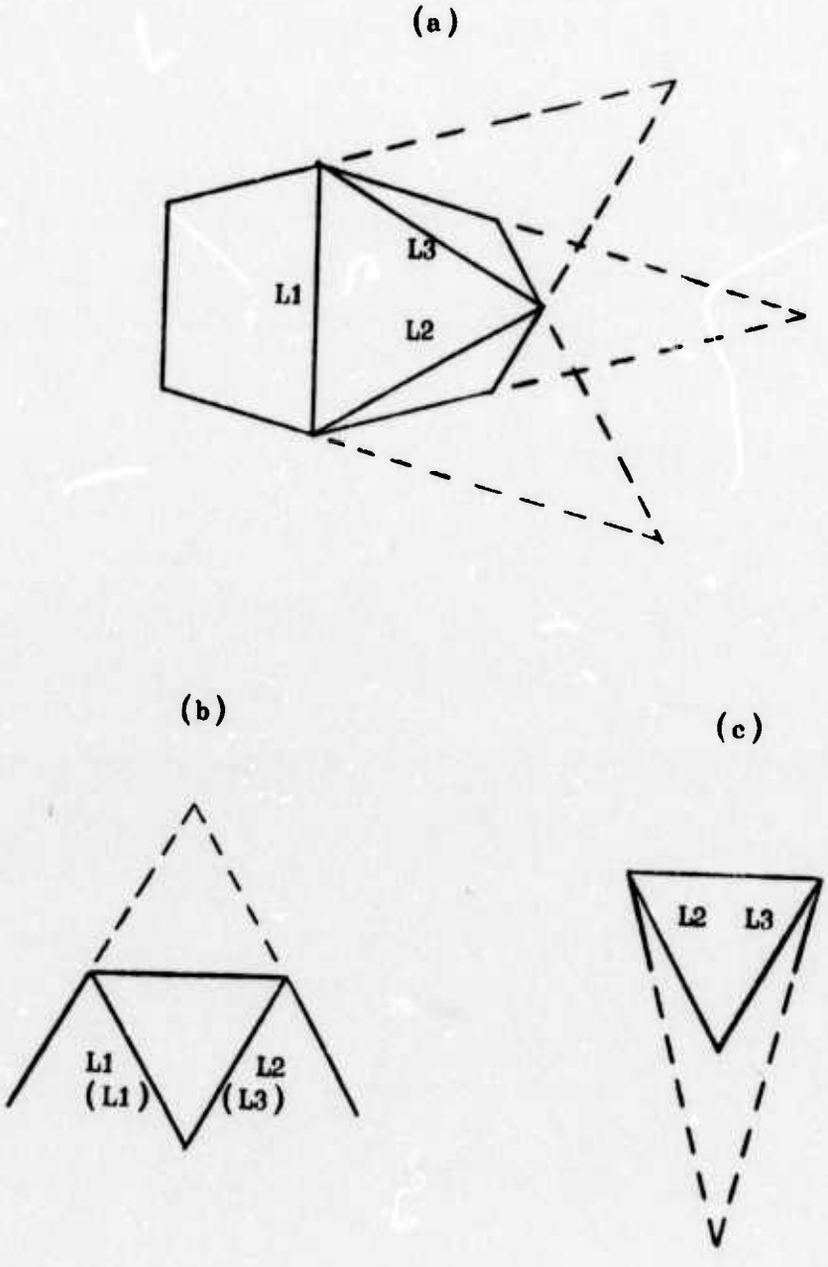


Figure 5.14

Triangle with exactly two similar CF:s

Figure 5.14 shows why the assertion does not hold for non-trihedrals. The lines L1, L2, L3 all have identical LF:s. The CF:s L1&L2 and L1&L3 are similar, whereas L3&L2 is in a class by itself. The discriminating item here is CF5 in the feature definition.

We now proceed to a discussion of projective invariance.

5.6 PROJECTIVE INVARIANCE

This is a basic idea behind depending on one single 2D prototype for each essentially different view of an object (cf. Section 12), and behind the construction of the features (and prototypes).

Definition:

Let C2 stand for the total class of two-dimensional perspective projections in which the same given faces of an object are visible.

We shall show that both LF:s and CF:s are projectively invariant over C2, given certain rather liberal constraints.

Referring back to the feature definitions (Subsection 5.2), it is easily seen that the following LF-items are invariant over C2:

PI1. LF2 and LF4 (constrained to trihedrals).

PI2. LF2+LF4 (not constrained to trihedrals).

PI3. LF3 and LF5 (in the case of strict equality, and then constrained to trihedrals). These bits are ignored for present purposes (used for degenerate views, see next section).

PI4. LF7 (under reasonable projective constraints, see below).

Thus the complete LF is projectively invariant over C2, with the constraints above. It follows that (with the same reservations) the following CF-items are invariant:

PI5. CF2 and CF3.

PI6. CF4.

PI7. CF5.

That is, the complete CF, as well, is invariant under the same conditions.

Proofs of essential points above:

PPI1. Otherwise a ray would have to shift over the extension of the parent line, which means that a previously seen face of the object would disappear. This is not necessarily true for non-trihedrals, as shown in Figure 5.15, where the top and bottom views require different models.

PPI2. By the same argument (also in the general convex case).

The reasonable constraints for PI4 are:

If the rays are parallel in space, the projections should still be within a liberal tolerance of being parallel. This is true where we are not too close to the degenerate case.

Otherwise the two rays, or their extensions, intersect somewhere (only the case where there really are two rays on the same side of the parent line is practically relevant), and we stipulate that the triangle with the base line as one side and the intersection of the rays as the opposing vertex, not extend beyond the plane of the observer (lens plane).

The latter case is illustrated in Figure 5.16. The shaded area shows (in 2D) where the observer must be situated in order for the projection to stay in C2. The object might be a truncated wedge, seen (by the

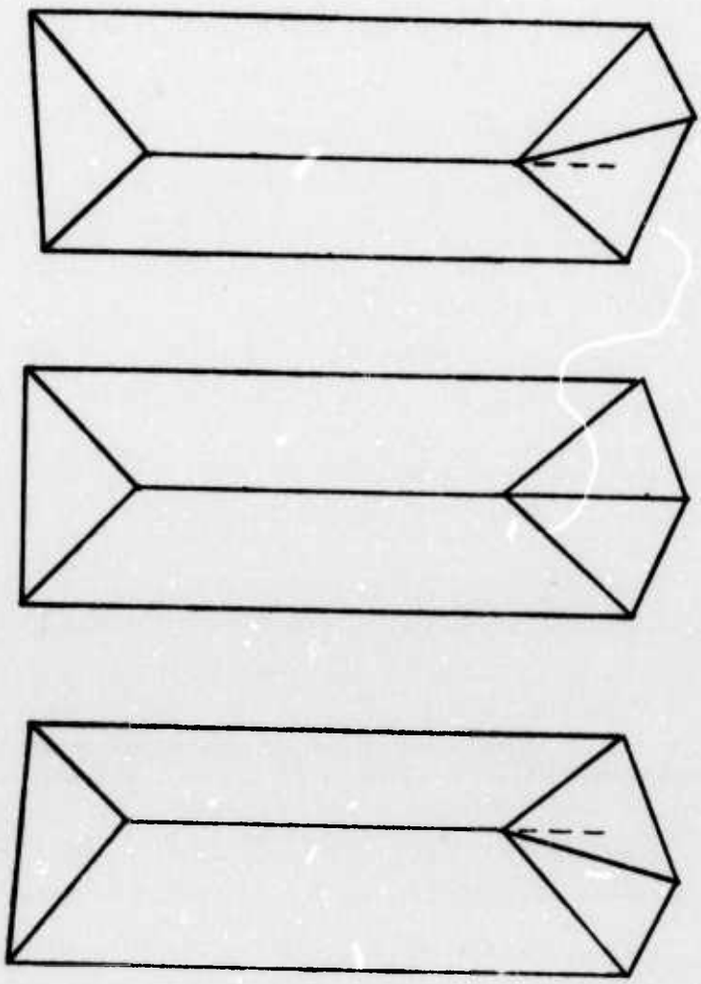


Figure 5.15

Trouble-causing non-trihedral

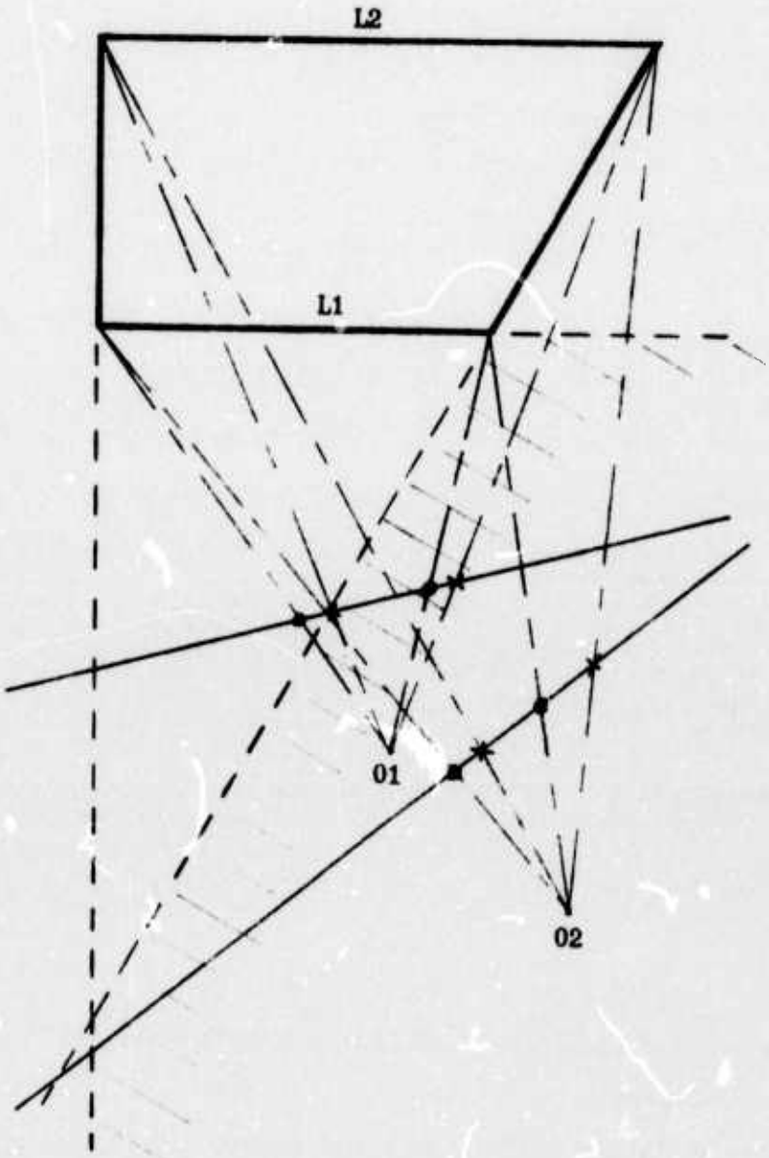


Figure 5.16

Projective constraints

observer, not the reader) from somewhere above and beyond. We note that for observer O1, the line L1 seems longer than L2, and thus that the connecting lines would seem to converge away from this observer. For observer O2, however, L1 seems shorter than L2, and the connecting lines seem to diverge, as they should, and as the two-dimensional prototype should indicate.

The condition for PI4 is always sufficient in order to preserve the convergence - divergence properties as expressed in the projection of the triangle. If we get closer to the object, divergence may degenerate into convergence (or vice versa), as we have seen. In the practical case this condition should very seldom be violated, and if so, then for "border-line" objects with edges deviating only narrowly from being parallel.

5.7 SPECIFICITY AND FEATURES

It may be of some interest to dwell shortly on the question of how much or how little information we would want a feature to contain, leaving aside the considerations of convenience in storage and handling, etc. The contention, naturally, is that the LF and CF contain the proper amounts, apart from being obviously convenient to handle.

Looking at the LF, there really isn't much more information around, given that we want to preserve the projective invariance. But two additional items might be included, namely:

LFA1. Connectivity of outgoing rays (ex. triangle).

LFA2. Convergence-parallelism-divergence for all combinations of rays on the same side of the base-line (not just the two opposing rays).

The reasons we do not want LFA1 are, first, that (in the scene representation) the connectivity of the rays may be obscured by other objects, in which case the features would not match - and, secondly, that there really is no need for it in the matching program, since that process uses the connectivity of the prototype as a template.

LFA2 might be more useful, mostly in the case of non-trihedrals. For example, in Figure 5.17 the fire-feature of the bottom line is the same for all three objects. Implementing LFA2 would be slightly painful, since we get a combinatorial amount both of storage and of handling. The second reason against LFA1 is still a good one here, complemented by the fact that the prototype acquisition program generalizes on parallelities, and that such information is used in the mapping process, as we shall see.

We would not want less, on the other hand, since all the information is necessary (to the present system) in order to provide power of discrimination and prediction (Section 6 - prototypes, and Section 9 - matching).

Similar arguments hold true in the case of the CF. In Section 6 we

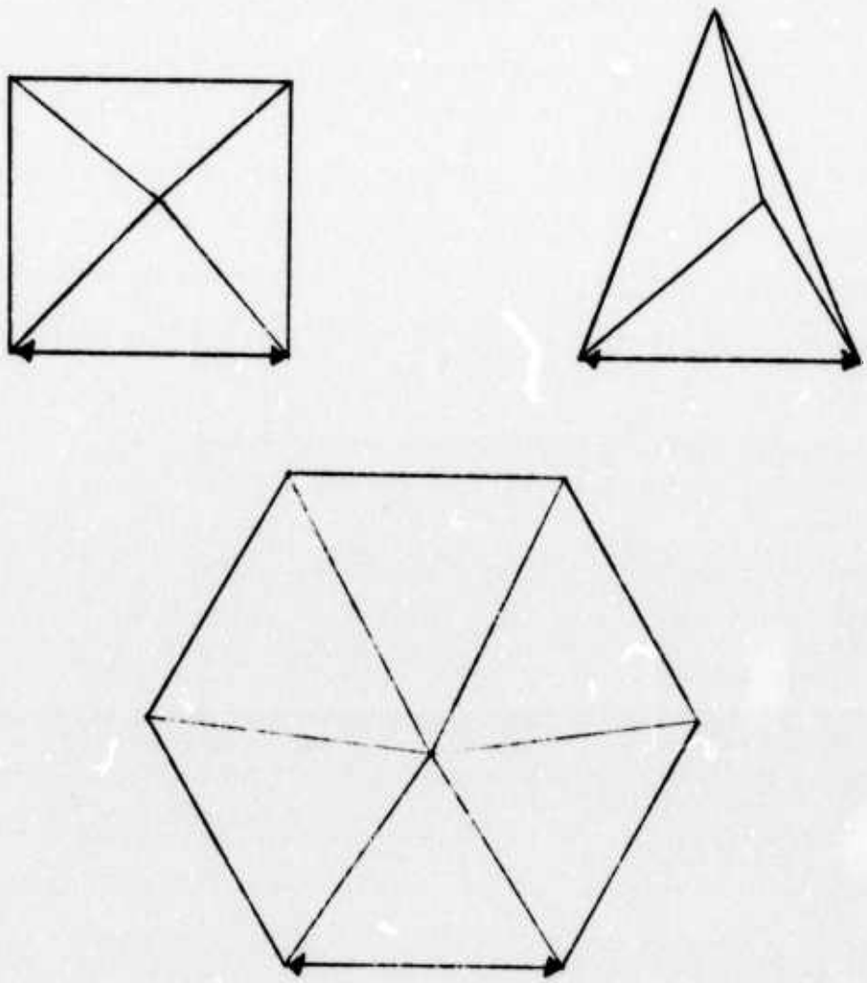


Figure 5.17

Secondary ray-constellations

shall deal with some additional aspects of features, such as their uniqueness properties as keys into prototypes.

6.0 PROTOTYPES

6.1 GENERALITIES

The handling of prototypes, like features, is fully generalized and automatic. This is true for acquisition as well as for their use in the matching process. The prototypes are perspectively consistent two-dimensional representations of views of objects in space. All objects are assumed planar-faced and convex.

We are not imposing a restriction to trihedral objects, but additional prototypes may be required here, as we have seen in Subsection 5.6.

Note that the restriction to convex objects has nothing to do with the basic structures of the features and prototypes. Those are quite general and would handle concave objects as well. No, the reason for this restriction is simply that concave objects give rise to an abundance of weird views (self-occlusions, vertex coincidences, edge alignments, ... you name it), each of which would require its own 2D prototype. They would also introduce difficulties in the form of partial matches.

Those circumstances would rapidly make the parsing strategy unworkable, due to overwhelming combinatorics. Experiments with an L-beam (two equally wide parallelepipeds "glued" together into an L) have borne this out.

Preceding page blank

An extension to concave objects may be based on regarding such objects as composed of several convex parts (cf. [Roberts] in Subsection 3.1, and Section 12).

The models are based on extended topological equivalence (including convergence- and parallelism-properties) and are therefore very general. Thus one single prototype is used to represent all non-degenerate views of parallelepipeds, including skewed ones. The final object classification, in an extended scheme, would take place in a context of three-dimensional models.

It would have been possible to use 3D models exclusively, with a projection-generating, feature-extracting program working directly on those, and using back-projections for purposes of matching. This possibility has been tested theoretically and, while conceptually appealing, would seem to introduce additional difficulties concerning generality and sensitivity to error. A discussion of related subjects can be found in Section 12.

6.2 INTERNAL REPRESENTATION

The internal structure of a prototype is based mainly on its constituent lines, and that information (below) is stored in a shared structure, pointed into by every prototype, since the number of lines varies between models.

The following basic items are stored for each model:

P1. Name (string).

P2. Number of vertices.

P3. Number of lines.

P4. Pointer into line-reference storage.

The following is the information for each line in a prototype (3 words of storage):

PL1. End-vertices.

PL2. Pointers to next lines (ccw.) at end-vertices.

PL3. Which side (if any) is part of the object contour.

PL4. Line-feature equivalence class (explained below).

PL5. Line parallelity class and length class (also explained below).

PL6. Line-feature identifier, and the LF itself for easy reference.

The lines are ordered (directed) the same as their line-features, and both lines and vertices are assigned an internal labelling. This makes PL1 through PL3 meaningful, and makes it possible to reference every element of a model.

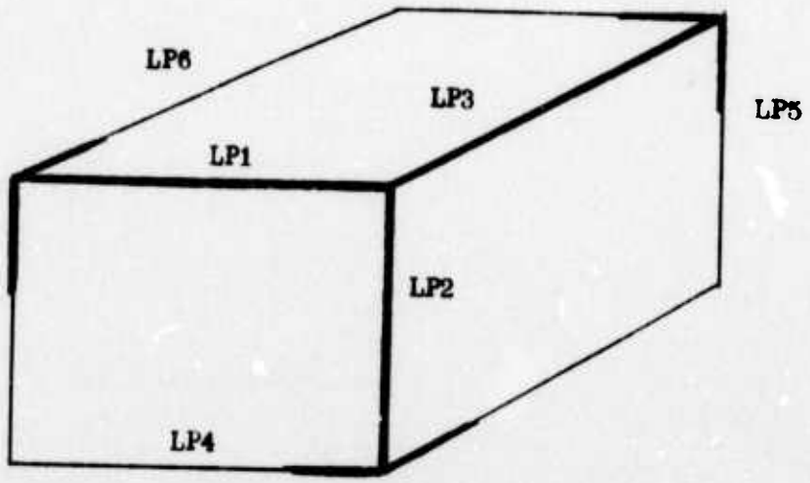
6.3 LINE-FEATURE EQUIVALENCE CLASSES

Let us for a moment contemplate the somehow familiar object in Figure 6.1. Let us assume that, somewhere in the scene, we have found the CF depicted at (b) in the same figure. The natural thing to say is: "Aha, it fits precisely on LP1 and LP3 in the model..". This is true, but the CF fits equally well on LP3 and LP2, or on LP2 and LP1. These are distinct lines in the internal representation of the prototype, as was noted above.

Looking at the figure, however, one realizes that all of those three initial mappings are equivalent, in the sense that the topology context (including parallelisms and convergence properties) is the same for each one of the three line-pairs. This can be clearly seen by turning the page around, and using in turn LP4, LP5, and LP6 as the bottom line of the object. Note that the different parallelepipeds in Figure 6.2 essentially (but for proportions) differ only in the angle of viewing. They are in the same C2 (Subsection 5.6).

This leads us to the conclusion that, having investigated the mapping

(a)



(b)

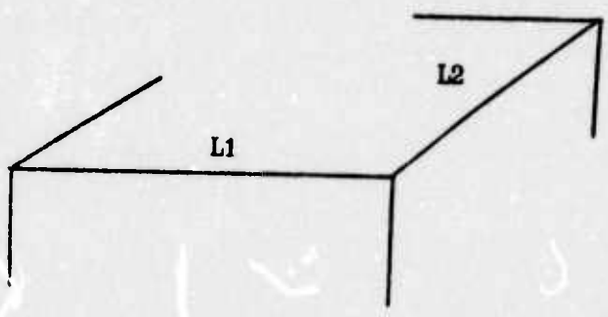


Figure 6.1

PAREP and equivalence classes

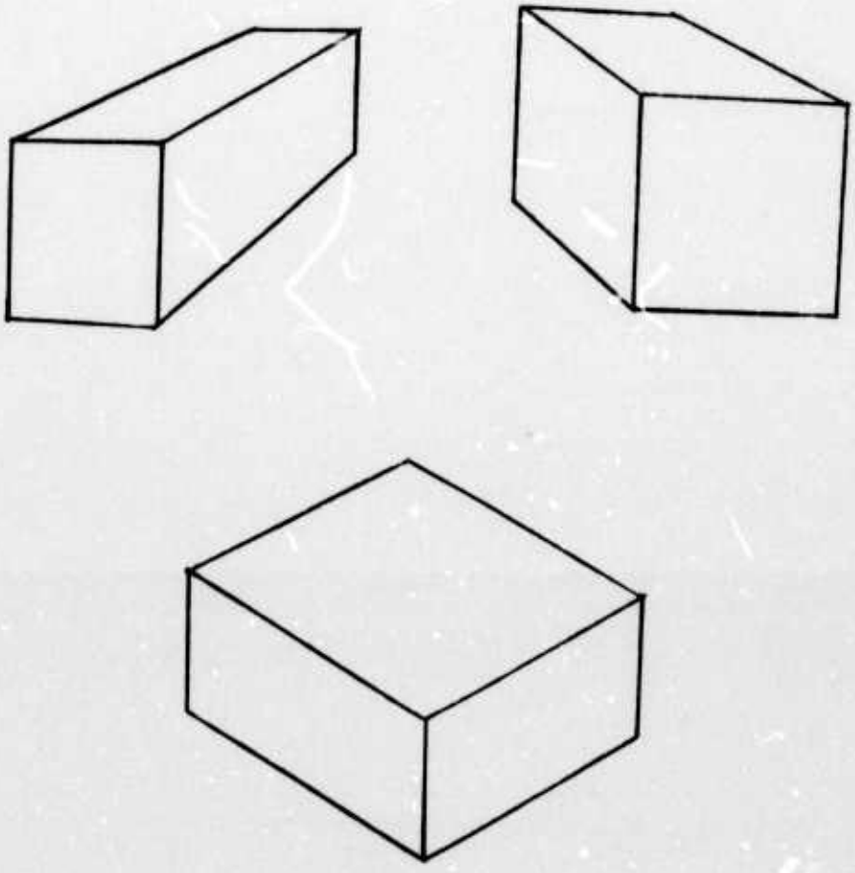


Figure 6.2

Same C2 - Same prototype

L1&L2↔LP1&LP3, there is absolutely no sense in bothering with L1&L2↔LP2&LP1, since the result will be no different from the first one.

We say that the line-features of LP1, LP2, and LP3 are equivalent, or in the same equivalence class. This is true also for the aforementioned CF:s, but that fact is not explicitly recorded in the prototype itself (since it contains no CF storage), only implicitly in the central feature reference pointer structure (Subsection 6.5). CF:s are not formally assigned equivalence classes. Of course the equivalence of CF:s is contingent on the equivalence of their LF:s, as defined below.

Note that the concept of equivalence class is meaningful only in the context of a specific prototype. We proceed now with the formal (recursive) definition.

DEFINITIONS:

Two lines (line-features) of a prototype are said to have the same equivalence classification if and only if the line-features are similar, and all lines attached to the two given lines (in the proper cc-wise order around the vertices, and in the direction of the LF) belong pairwise to the same equivalence-classes.

Two line-pairs (compound features) are said to be in the same equivalence class if and only if their respective compound feature words are similar, and their constituent line-features, taken in the order of the internal orderings of the CF:s, belong (pairwise) to the same LF-equivalence classes.

The following algorithm is used in the prototype analyzer to determine the equivalence classes for line-features.

ALGORITHM:

A. Give the lines an initial assignment of tentative equivalence classes, a different class for each different line-feature type, so that the initial condition (feature similarity) is satisfied.

B. For each equivalence class, EQ, the first encountered line is now assumed correctly classified. Go down the list of other lines belonging (so far) to EQ, checking whether they conform to the definition, using the original line as a template. If a line does not conform, make a note of this, but do not at this stage change the classification.

C. If there are no changes noted, exit. Otherwise change all marked lines, so that all such lines of a given EQ are assigned the same, new, equivalence class. Iterate from B.

Note that we cannot effect changes as soon as the need is seen, since we might encounter a situation where such action would partially change some EQ assignment, thereby obscuring the fact that some pair of differently grouped lines should really have the same classification. As a matter of taste, we could make the change between equivalence classes, but it doesn't make much difference, and the present algorithm is convenient for programming reasons.

We now proceed to prove the correctness of the algorithm, and also that it provides the minimum-spread such classification, i.e. that two lines are classified differently if and only if they do not belong to the same EQ, according to the definition.

PROOF OF ACCURACY OF THE ALGORITHM:

From the fact that step B of the algorithm analyzes the complete prototype, testing the EQ-classifications for all lines according to the recursive definition, it follows that those classifications are in accordance with the definition, when the algorithm is exited. Otherwise step B would be reiterated.

On the other hand, all lines with an initial assignment to an equivalence class are assigned the same new classification if and only if they do not conform to the first line of that class. Furthermore, the changes are performed at the same time, just before iteration, and do not influence the conformity tests in step B. Thus it is impossible to exit with two lines classified differently, unless they should be.

6.4 PARALLELITY AND LENGTH GENERALIZATIONS

The prototype analyzer generalizes on two things particularly and explicitly (besides those generalizations inherent in the features), viz. parallelity and length, in the following restricted sense:

G1. Two lines in a prototype are said to be in the same parallelity class if and only if the smallest difference between their angular arguments in some direction is less than some given limit, currently 5 degrees.

G2. Two prototype lines are said to belong to the same basic length class if and only if they are in the same parallelity class (length class = parallelity class). However, we allow two length-categories, one long and one short, within each length class. Any line (within some length class) will be assigned to the longer category if and only if it is longer than 1.25 times the length of the shortest line in that length class.

The chief reason for the use of parallelity classes is prediction, where we may have to know the approximate direction of a missing line in order to insert a tentative one, or the direction we expect a new line to have, in order to be able to discard one that deviates too much. This is not always possible on the basis of the line-feature data alone (the only feature used throughout the mapping), since the parallel lines may sometimes not be simply connected. It is also convenient for easy referencing.

6.4

There are two basic reasons for the introduction of length classes. The first one is that knowing the approximate length of a line, we may be able to quickly decide whether to believe in it, or to look for an extension, or if it seems necessary to divide the line and use only part of it.

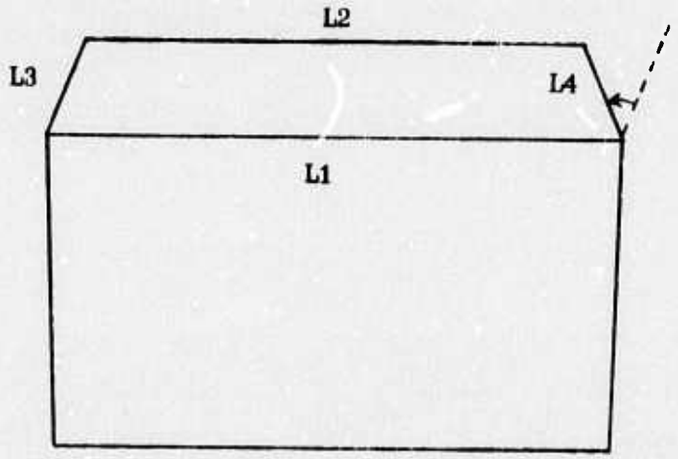
The second reason is that it gives us a more tangible hold on perspective, since perspective deformations have less effect on relative lengths within parallelity classes than they have on angles. Figure 6.3, part (a), shows this clearly. The lines L3 and L4, while parallel in space, have an angular difference of about 45 degrees, whereas the effect on the relative lengths of the parallel lines L1 and L2 is much slighter (somewhat awkwardly expressed), L1 being about 1/8 longer than L2. Thus the relative lengths of L1 and L2 would not refute the assumption that L3 and L4 are parallel, which the prototype demands. Using the angle alone, we would have to set the discriminator very liberally, thereby likely introducing erroneous assumptions elsewhere.

The truncated wedge in Figure 6.3, part (b), indicates the reasons for introducing length sub-classes. We are assuming that we will not be dealing with objects that would necessitate more than two such categories.

We shall sometimes talk of "equality-classes" as a collective term for these generalizations.

The concepts above (reasons for -, use of -) will become clearer further

(a)



(b)

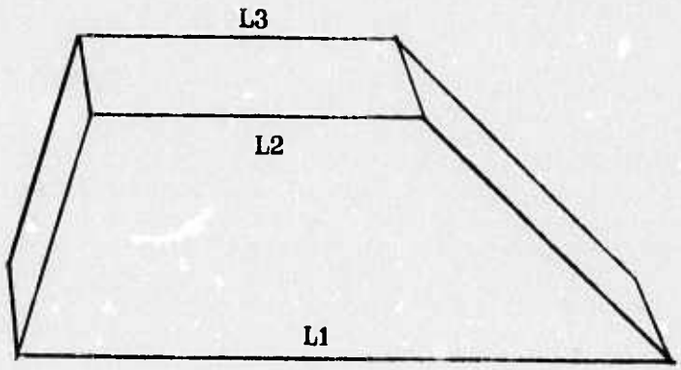


Figure 6.3

Prototype line length-classes

on (Section 9), in dealing with the matching from scene elements to prototypes. We shall now briefly return to the feature structure.

6.5 CENTRAL FEATURE REFERENCE STRUCTURE

The following is a description of how the feature table is built up, with reference to prototype access. Use is made here of the concept of equivalence class, so that redundancies are avoided.

Figure 6.4 shows the details of this central feature reference list structure. This storage is a complete ordered array of all features found in the prototypes, augmented by pointer structures for references back to the prototypes. The prototype analyzer ascertains that there is exactly one reference from each different line-feature to each model that contains that specific LF, and to some line belonging to each equivalence class of that LF, within the prototype.

In the case of CF:s, we make sure there is exactly one pointer to each line in the pair of the CF, with similar restrictions to avoid redundancy.

The reference list also contains pointers to all CF:s encountered in the scene at any given time of analysis. Therefore the parsing program simply goes down the lists, exploring feature matches in order of decreasing feature complexity, essentially investigating all initial mapping possibilities.

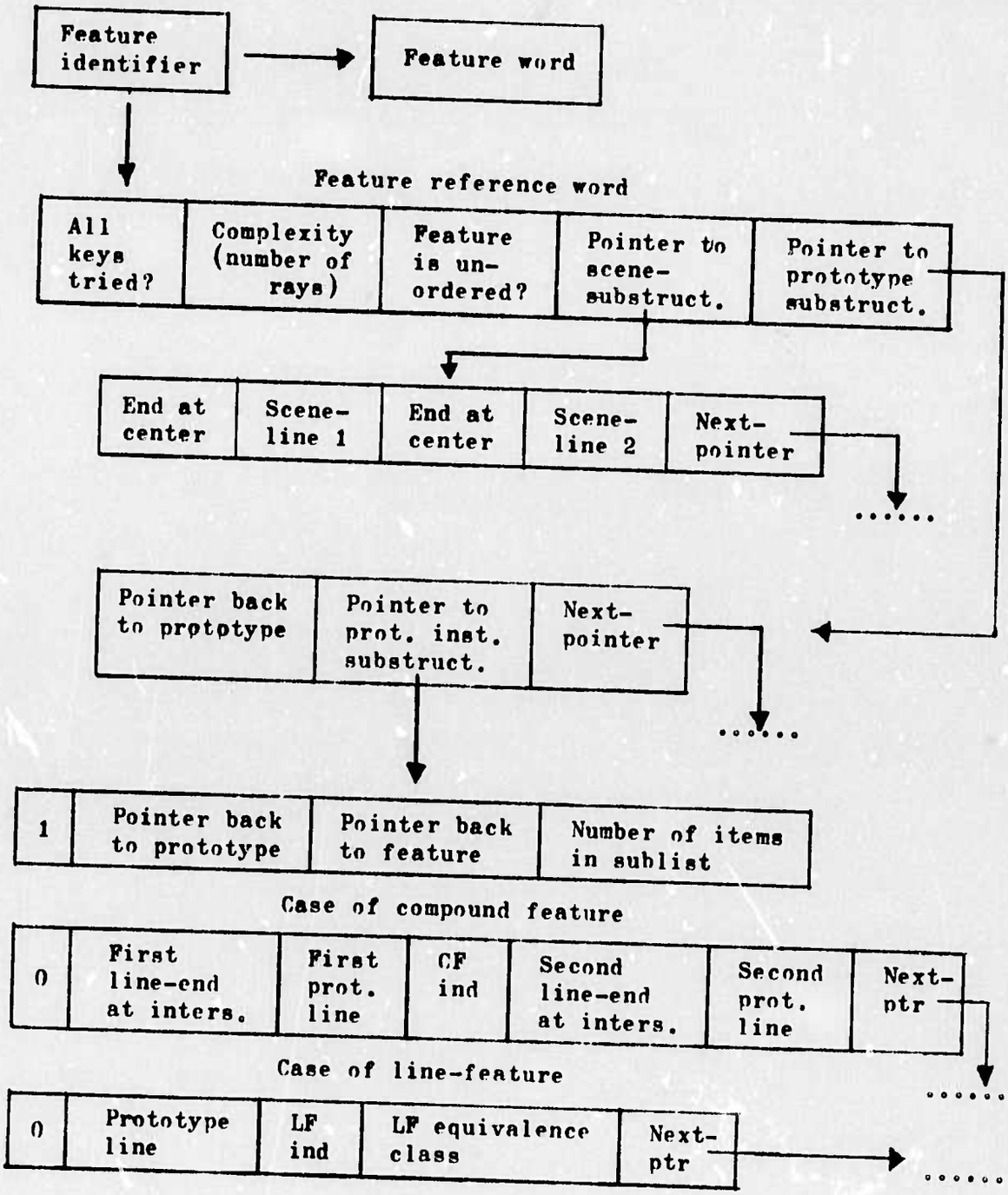


Figure 6.4

Central feature reference storage

We shall return to these subjects in the context of the parsing process, Section 8, which among other things describes the feature extraction over the scene.

6.6 PROTOTYPE ACQUISITION

As has been pointed out, the acquisition (or "learning") of a new prototype is fully automated, and all prototypes are treated exactly the same. The following is an account of the steps in the input of a new prototype.

IP1. Input perspectively consistent line-drawing.

IP2. Analyze this line-drawing, using the pre-processing package.

IP3. Call the prototype analyzing program.

IP4. Flush the line-drawing and associated data-structures (created in IP2).

Main actions performed by the prototype analyzer are:

PA1. Classify constituent lines in terms of line-features. If heretofore unknown features are encountered they are added to the central feature list.

PA2. Create compacted topological data-structure for the model.

PA3. Find LF-equivalence classes, parallelity classes, and length categories.

PA4. Update LF pointers in the central reference list, so that it contains one reference to this prototype (and a line) for each combination of LF and LF-equivalence-class.

PA5. In parallel with PA4 find CF:s, and update the central feature list as in PA1, and also update pointer structures similarly to PA4 (Subsection 6.5).

The following are some comments to clarify steps above.

In IP1 the line-drawing is given by providing (from the console or via a file) the end coordinates for all participating lines. Care must be taken to obtain approximate parallelisms where such are desired, and to avoid them where unwanted. By "perspectively consistent", we mean that spatially parallel lines should be adjusted length- and angle-wise by

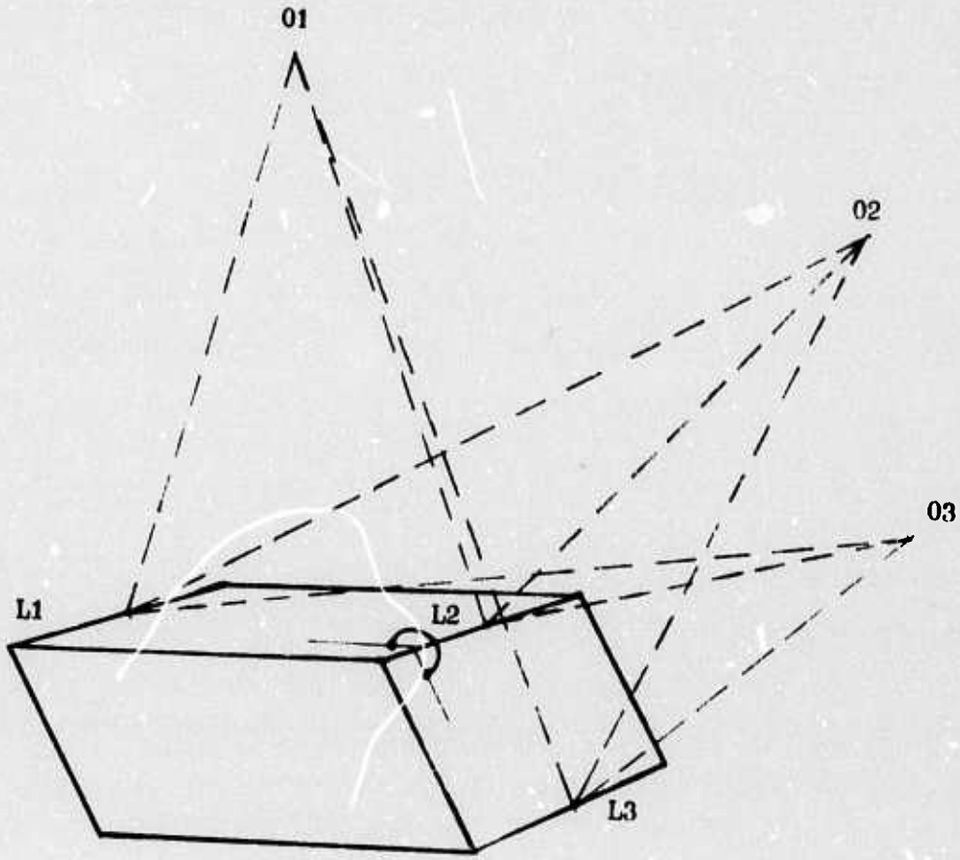
some small amount in order to indicate a perspective deformation, since that concept is used in the matching program (Section 9) (and only there).

Sometimes we cannot generalize on such perspective deformations, namely if those object faces (that contain the parallel lines) form an exterior angle of less that 270 degrees, in which case we get a dependence on orientation. Figure 6.5 demonstrates this state of affairs, in the case of a skewed parallelepiped.

Such line-drawings could be generated automatically in a full-fledged 3D system, as indicated in Section 12 (future possibilities).

Step IP2 entails finding the vertex connections and setting up the normal cross-reference data-structure. Such things are treated later, in Section 7.

After the learning of a prototype, all that remains is the internal representation, not the line-drawing. This data-structure (for all current models) may then be conveniently saved on auxiliary storage. We may thus have different sets of models, which can be used easily and at will. One may conceive of a future system that makes some intelligent use of such different sets of prototypes, trying a new set if the current one seems to yield unsatisfactory results. It would to some extent be able to accomodate itself to the surroundings. However, there is no use for such a scheme in the present system, but possibly in a more sophisticated one, where we utilize three-dimensional models and



Skewed parallelepiped resting on table.

Orders of apparent lengths of L1, L2, and L3 for the observers at 01, 02, and 03, respectively:

01: L1-L2-L3 02: L2-L3-L1 03: L3-L2-L1

The observers are all thought to be in a plane parallel to a plane through the center points of L1, L2, and L3.

Figure 6.5

Orientation dependent perspective deformation

have access to depth information in the analysis of the scene (Section 12).

Another possibility is to have the program learn new prototypes by "consistent encountering", i.e. by finding something new a sufficient number of times to conclude that it probably is some object it should know about. Such a scheme is nice because it is more general, but it is also more error-prone, since we would not necessarily encounter perfect (enough) instances of the object projections.

In an extended scheme (3D) the prototypes would be given by the end-coordinates of their edges, and the acquisition program would generate all different views of the object in question, creating a new 2D model whenever the current projection does not map onto any of the existing 2D prototypes.

The next two sub-sections deal with the currently used set of models, and will provide some discussion of the extent to which objects can be conveniently and unambiguously represented through the prototype and feature schemes given here.

6.7 CURRENTLY USED PROTOTYPES

In this subsection we refer to Figure 6.6, which provides a set of the most useful (and realistic) models. The most often used prototypes are given in the following table (for a definition of "degenerate", see Subsection 6.8).

M1. PAREP: Parallelepiped (non-degenerate).

M2. WEDGE: Wedge (non-degenerate).

M3. DPAREP: Parallelepiped (degenerate).

M4. DWEDGE: Wedge (degenerate).

M5. TWEDGE: Truncated wedge (auxiliary model).

In other words we have four 2D prototypes, which represent all possible views of our two different objects (not counting the TWEDGE). The choice of objects was based on their simplicity and regularity. Of course, one might want a more varied set of models, such as a tetrahedron, truncated objects, etc. The truncated wedge has been used from time to time, experimentally. It is not currently an active prototype.

The fact that one of the models (the PAREP) may be thought of as composed of two instances of another (the WEDGE) tests the

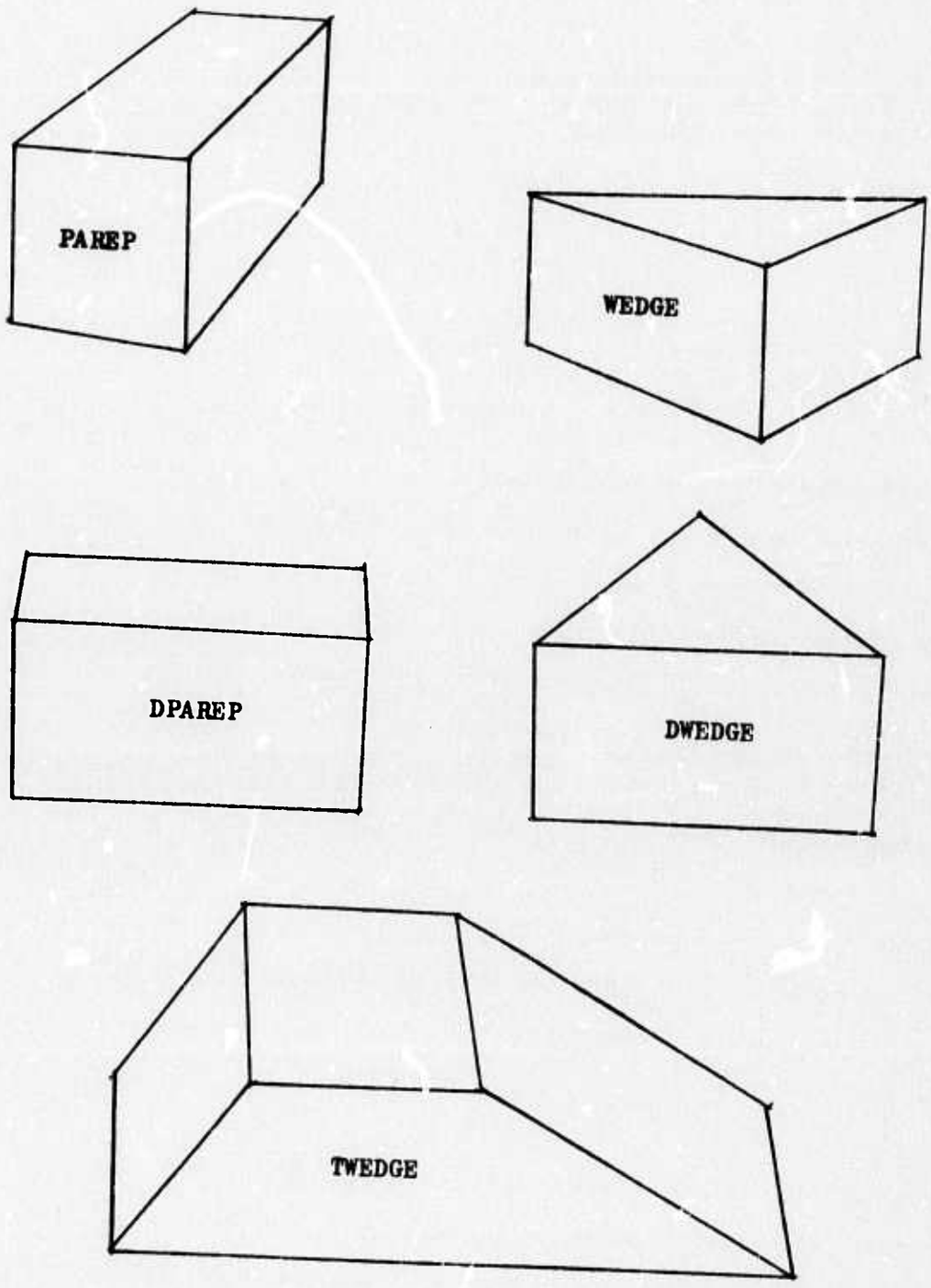


Figure 6.6

Current and auxiliary prototypes

discriminatory powers of the system, since it introduces partial matches. We shall get back to that topic later, in Section 9 and Section 11.

6.8 DEGENERATE VIEWS

As we have seen (Figure 6.6), the prototypes contain representations of degenerate views as well as "normal" ones. A degenerate view is here defined as one in which there is no vertex where more than two side-regions meet. Usually such a view is one where, for that orthogonal projection which shows the same sides of the object, rotating the object a small angle around some axis would change the topology of that orthogonal projection. Note that with suitable projective constraints (Subsection 5.6) there is always such an orthogonal projection.

We shall use the term "perspectively degenerate" in the case where a similar rotation would change the topology of the perspective projection. We shall sometimes use the obvious abbreviations D-view and PD-view.

Thus (a) in Figure 6.7 shows a degenerate parallelepiped and wedge, whereas (b) represents perspectively degenerate views of the same objects. Note that the term degenerate is used somewhat inconsistently with its usual meaning in cases like the wedge. It was chosen for convenience.

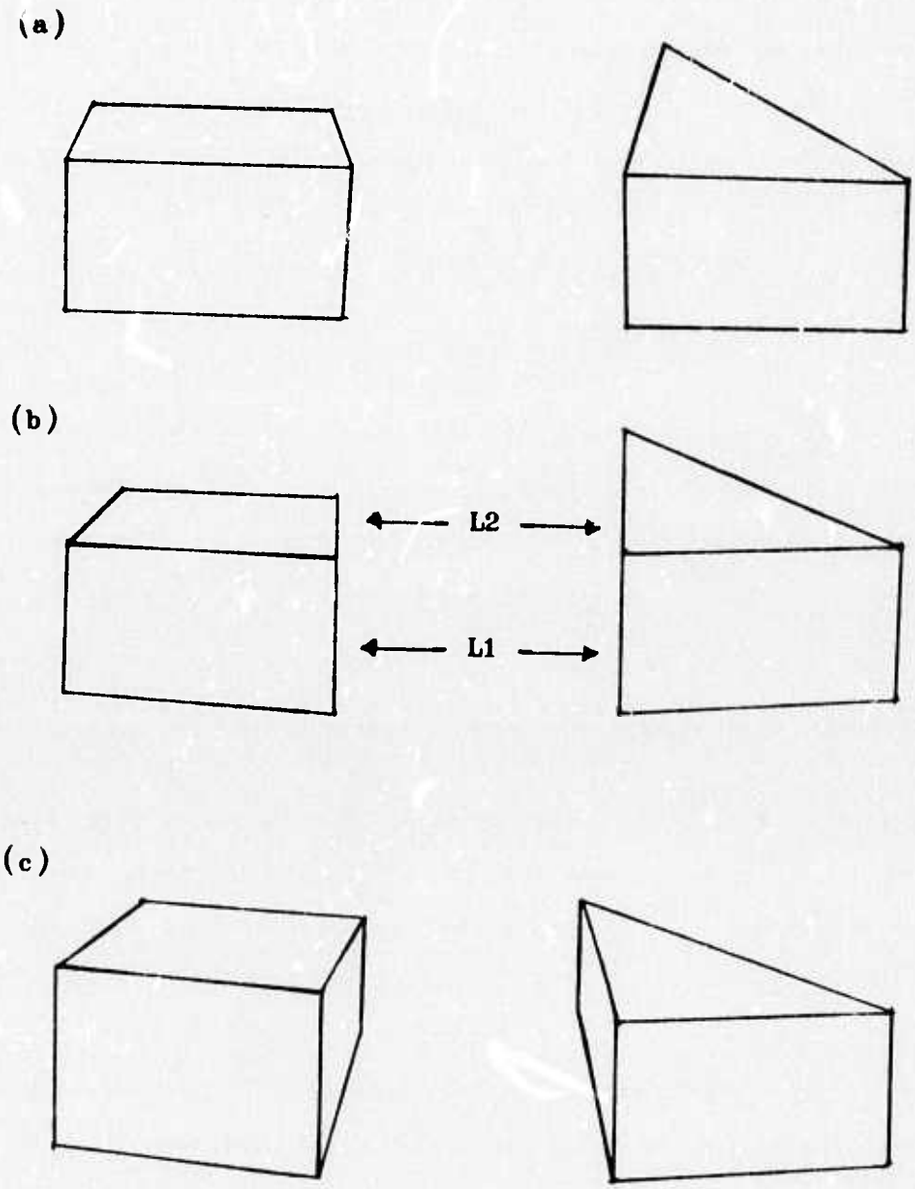


Figure 6.7

Degenerating views

So, in the present system, degenerate views are represented by degenerate models. However, we cannot do the same for perspectively degenerate views, since in those cases (cf. L1 and L2 in the same figure) we do not often find the initial lines representing degenerate planes unbroken. On the contrary, they are often split into two or more parts which form small angles with one another. This often makes it difficult to decide whether we are dealing with a PD-view or not.

As an added attraction, we often get views like those in Figure 6.7, part (b), due to occlusions. In most cases, however, such an occluding object should be better matched to some prototype and thus disappear, leaving us with a partial mapping (part (b), with L1 and L2 gone).

Therefore PD-views is one of the problems in the present system, as indeed they would tend to be (I suspect) in any vision system dealing with the real world. They have to be regarded as special cases of D-views. On the other hand, we want to be able to pick up the marginally non-degenerate cases, as indicated in part (c) of the same figure.

What makes the problem hard is partly that a very slight change in the data may result in a dramatic change in topology. The other unfortunate circumstance is a consistent lack of helpful edge-information in such areas, due to their narrowness. This makes it hard to verify predicted line-elements. A slight amount of ad hoc -ery has been necessary in order to detect these cases and channel them into the proper prototypes. This is done by channeling border-line instances (where an outer angle of an LF is between 180-Alpha and 180 degrees, Alpha currently being set

to 7.0) into the degenerate case, rather than trying to complete them as regular, almost degenerate objects. The reason for this is partly very practical, since some subroutines for intersections, collinearities, and the like, get fouled up when dealing with a region that has been squeezed almost into a line. Nevertheless, there is a global switch to enable this scheme.

6.9 REPRESENTATIONAL AMBIGUITIES

It may be interesting to make an assessment of the extent to which objects can be adequately and unambiguously represented through the features and prototypes suggested here. That is, are there objects for which the parsing program, or rather the prototype matching program, might mistake one object for another?

Of secondary importance is the uniqueness of the initial line-mappings provided by (primarily) the compound feature and (secondarily) the line-feature. The reason this is not crucial is that the matching program has the full power of decision and will give low marks to bad mappings.

Let us look at the last question first (see Figure 6.8). The line-feature, applied on L1 in the fig., will put all three objects in the same class. The compound feature, applied on L1 and L2, will be able to distinguish between (a) and (b) but not between (b) and (c). However, the objects with which we are dealing are usually not as complicated as

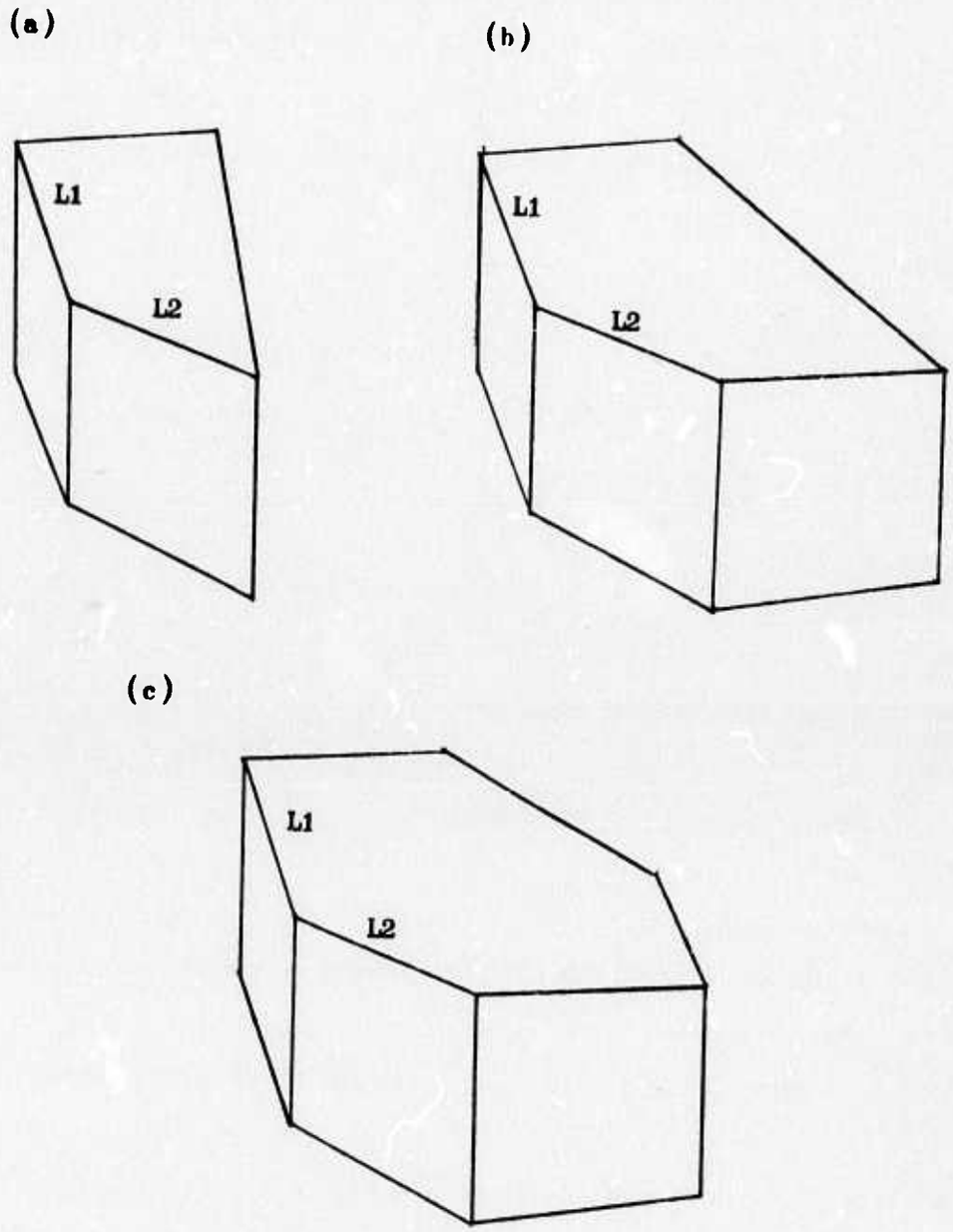


Figure 6.8

Keys and ambiguities

that. The following table (over the four mostly used prototypes) demonstrates the performance of the compound feature in terms of uniqueness as initiator into mappings:

Total number of CF:s	30.
CF:s mapping into 1 prototype only	25.
CF:s mapping into 2 prototypes	4.
CF:s mapping into 3 prototypes	1.

This shows that for the most commonly useful (uncomplicated) objects, the compound feature is quite an accurate guide for mapping initializations. Of course, degree of uniqueness is directly proportional to complexity (number of rays), and therefore the line-feature is much less suited for mapping initializations.

Now to the main question:

How similar do two object projections have to be, in order for their prototype - feature representations to be potentially subject to confusion?

Clearly, to start with, the topologies must be the same. Furthermore every pair of corresponding line-features has to be similar between the two projections, so that, at every vertex and on both sides of the extended base-line, the topologies must agree. Angular convexities must also agree, for all line-junctions. Parallelities and relative lengths of parallel lines must agree (within tolerances), not only as given by the limited reach of the line-feature, but also as recorded in the parallelity- and length-class items, which reach over the entire model.

We may therefore conclude that ambiguities in the prototype representation of 2D projections are introduced only in terms defined by our tolerance levels for parallelity and length-quotients.

Figure 6.9 gives an example of such ambiguities. Clearly, the models may be constructed (and analyzed) with any desired levels of tolerance, but the crucial issue is how well we (the program) will manage to distinguish between them in the parsing process.

Here is one case in which the perspective information (as indicated in the prototypes) may be helpful. Thus in Figure 6.9 we may be able to distinguish between (a) and (b), or (b) and (c). However, (c) might be a perspectively deformed version of (a), or it might be another model (a truncated wedge, on its head, for instance).

This is a case that should not be likely to arise in practice, and where (if it did) we would be compelled to rely on 3D knowledge for the decision.

This concludes the sections on features and prototypes, and before continuing with the related topics of mapping and parsing I shall now briefly describe the nature of initial data and necessary preprocessing stages.

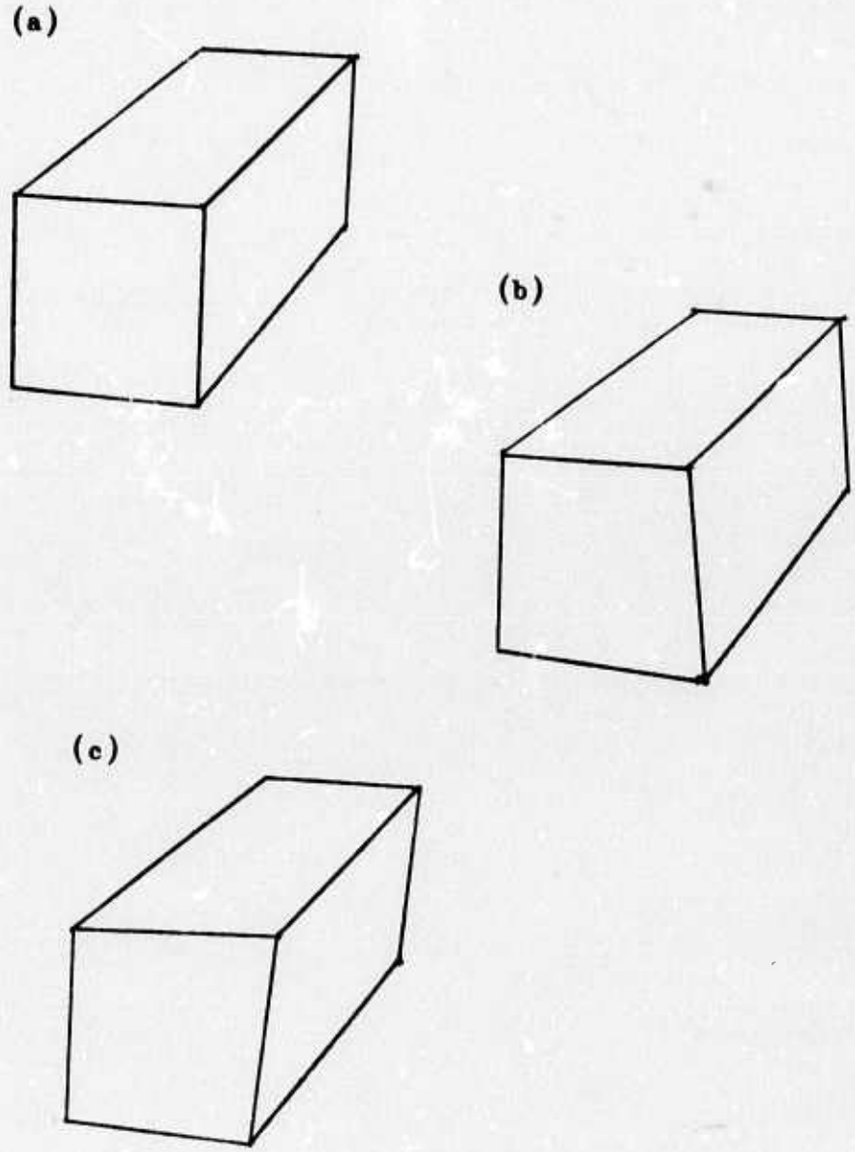


Figure 6.9

Potentially ambiguous representations

7.0 PREPROCESSING

7.1 INITIAL DATA

The initial input consists of an array of two-dimensional coordinates for locations of intensity discontinuities in the TV-image. Those points are called edges throughout this paper. That total input is hereafter called the edge-drawing. Figure 7.1 demonstrates the character of such initial edge-drawings.

The edge-follower is part of the Stanford Hand-Eye System. It was coded mainly by Karl Pingle [Pingle & Tenenbaum 1971]. It uses Tenenbaum's accommodation routines [Tenenbaum 1970] and the powerful edge-detecting operator created by Manfred Hueckel [Hueckel 1971 & 1973]. I shall not attempt to describe the operation of the edge-follower in any but the following extremely broad terms.

The edge-operator consists of a variable size, approximately circular matrix which, applied over some small area of the TV-raster, utilizes a number of elaborate mathematical functions to obtain (basically) the location of the edge, the intensity gradient vector, and the brightness difference. Figure 7.2 shows an ideal edge, its intensity profile, and the resulting operator output.

The edge-extraction is performed on a 333*256 matrix of intensity

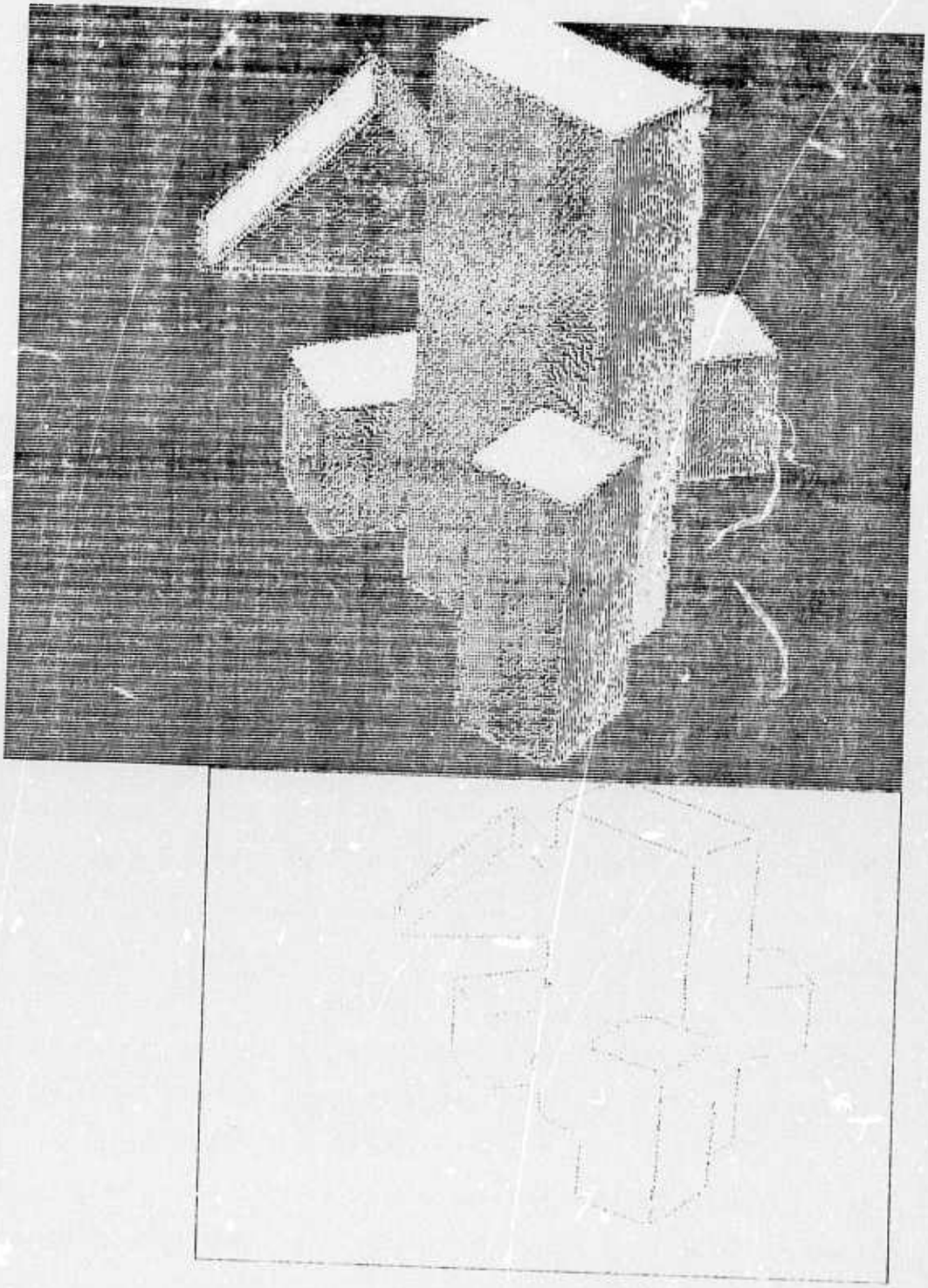


Figure 7.1

Initial input: Edge-drawing

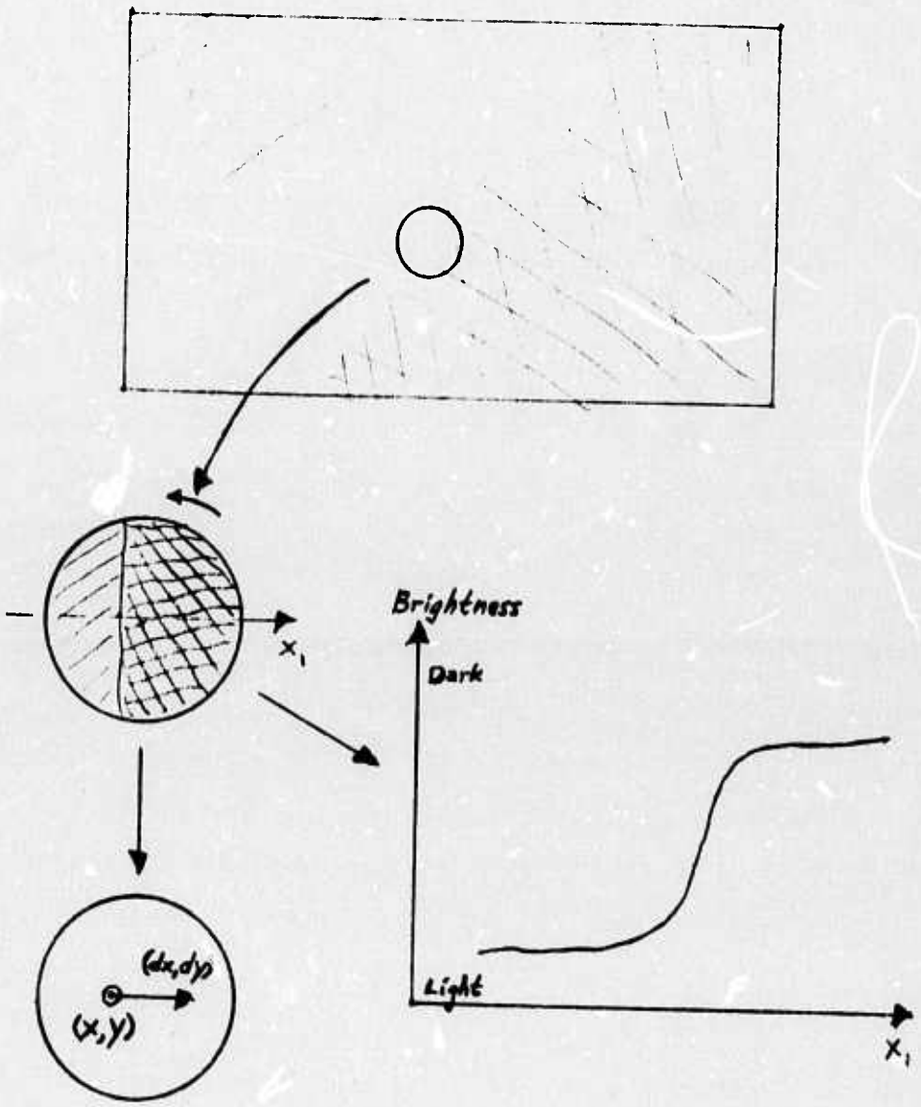


Figure 7.2

Edge-detection

values, each of which has 4 or 6 bits of information. A single TV-scan results in 4 bits, but 6 may be obtained by combining several scans at different intensity ranges.

The edge-follower makes a coarse scan over the picture until it finds an edge, which it subsequently tries to follow until a closed curve is found. It contains a line-fitter which it uses to obtain some idea of the locations of the vertices in the scene. A closer edge-scan may then be performed in some area around each of those vertices, so that other lines may be detected. Since this may sometimes lose (due to glare, shadows, adverse lighting conditions, etc) there is another mode available, in which a complete scan is performed on the inside of all closed regions found previously. The program accomodates the sensitivity of the TV-camera as it proceeds, so as to be able to see better in the local area of current interest.

Alternatively we may work on stored TV-matrices, in which case accomodation is by definition impossible, and where the quality of the edge-drawing becomes lower (as a rule), even if 6-bit intensities are used.

Whatever the case might be, as the next step in the processing of the picture, the original edge-data is transformed and sorted before we start the line-abstracting phase.

The transformation replaces each edge-point and gradient vector by an edge-pair (see Figure 7.3, (a)), so that the direction of the local

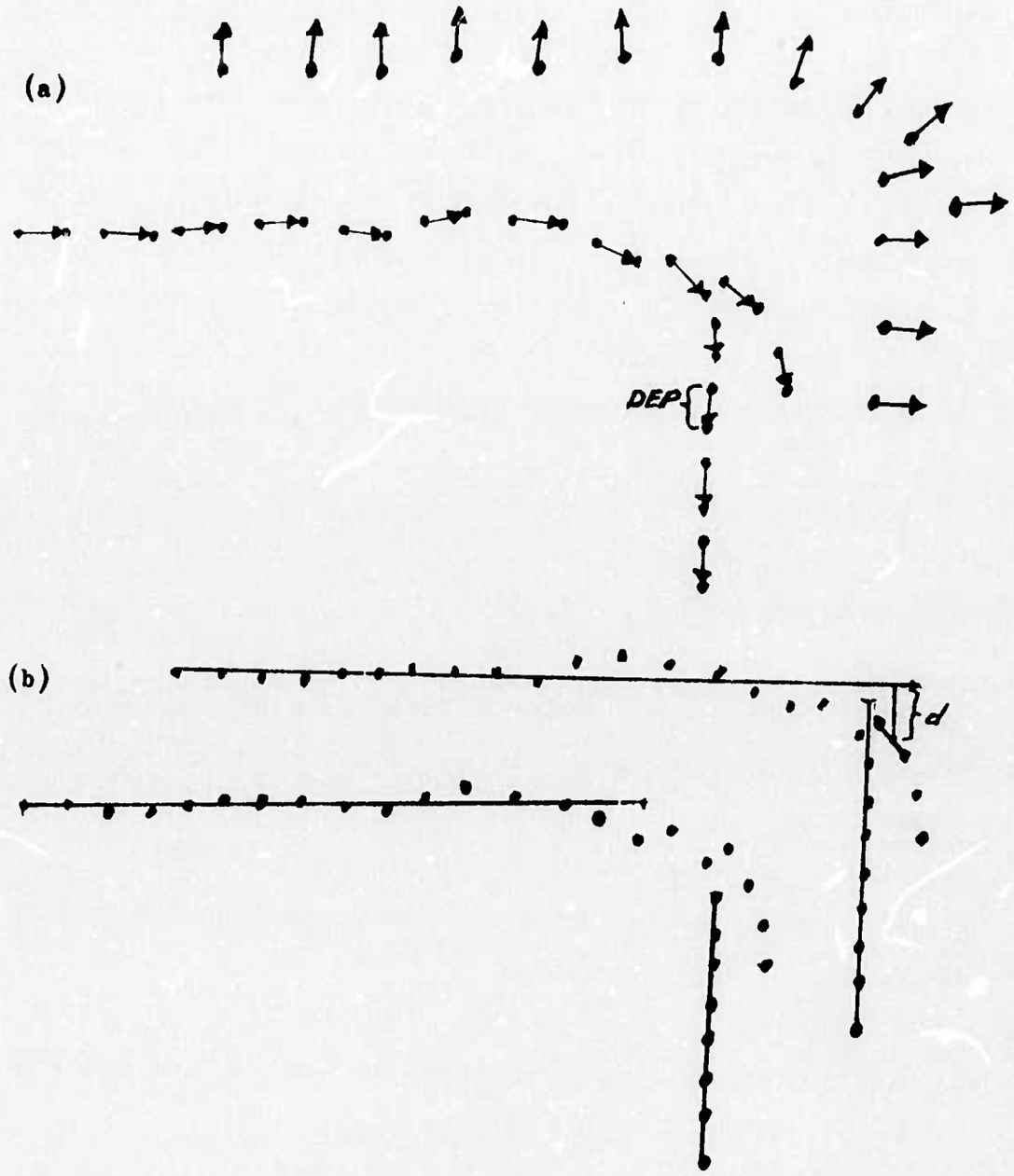


Figure 7.3

Initial input - edge-data

intensity-discontinuity is from then on implicit in the vector formed by each pair.

The sorting program creates a linkage among the edge-pairs to ensure that each edge-pair is in the proper context in terms of closeness to other pairs, and in terms of the angles of the pair-vectors. In other words, the list of pairs is ordered in a way conducive to extracting the best possible lines. The input data already to a great extent conforms to such an ordering, but it is not satisfactory in areas near vertices, or in other regions with complicated patterns.

7.2 ABSTRACTION OF INITIAL LINES

Looking at Figure 7.1, our (human) vision system tends to abstract shapes or objects from the data. It is unclear (to me) how big the chunks of abstracted information are, but it seems (judging from my own experience) that we intuitively perceive lines where the picture contains straight arrays of edges, and that the patterns of those lines are interpreted in meaningful ways.

Be that as it may: Line-extraction is the first point on the agenda for the present system.

The line-extracting program attempts to fit lines within the connected subsets of edge-pairs resulting from the sorting phase, and it uses an exact least square method for the line-fit.

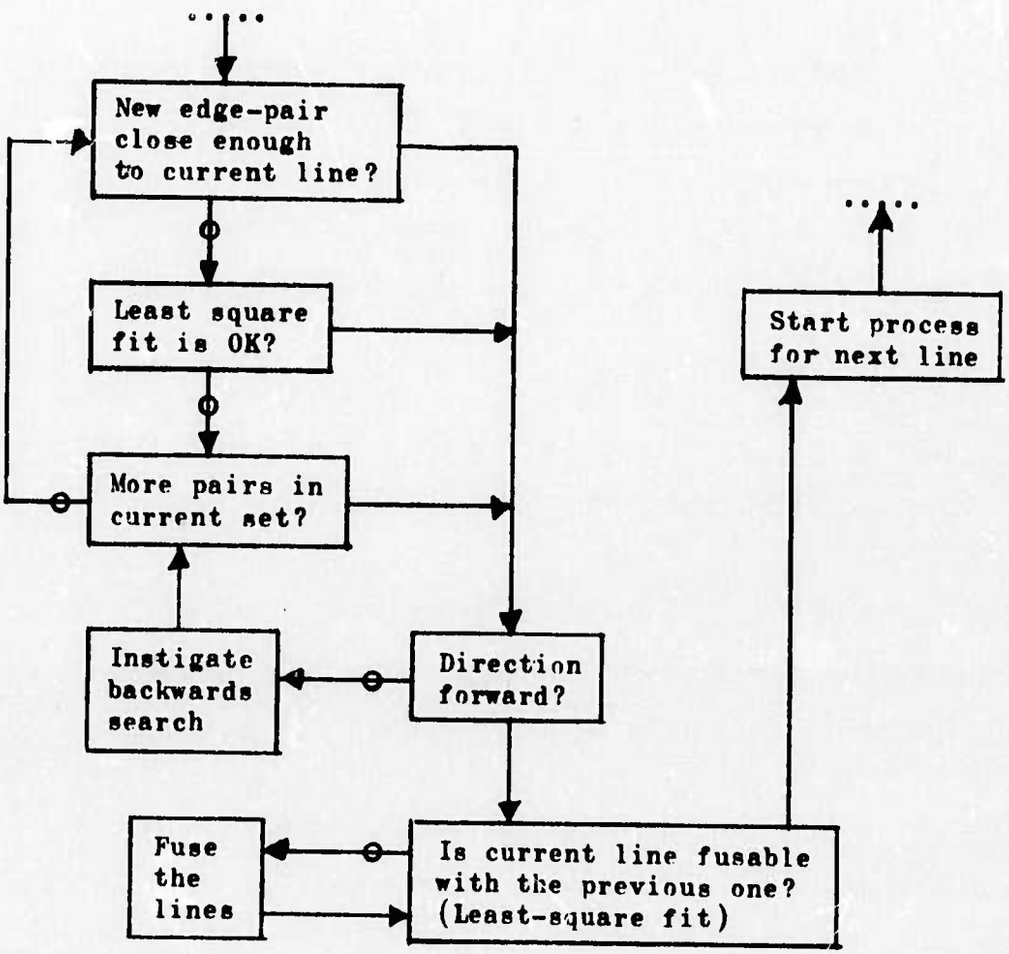


Figure 7.4

The line-extracting algorithm

Figure 7.4 gives the flow of the line-finder. A couple of things are worth noting here.

1. New edge-pairs are tested for closeness to current line (before least-square fit) and rejected (line stopped) if not close enough. This prevents wrapping around corners, as Figure 7.3 also demonstrates (part (b)). The least-square fit itself, for long lines, is not sensitive enough here.

2. When we get no further in one direction, we try extending the line at the other end, in the same kind of process.

3. When both directions are exhausted, we try merging the present line with the previous one, iteratively, before starting on a new line.

After all possible lines have been created, we finally clean up the picture, removing lines that are based on an insufficient number of edge-pairs (parameter), and shrinking each one of the rest of the lines by an amount proportional to the quantity DEP in Figure 7.3, limited by an amount proportional to the length of the line. This is done in order to clean up around the vertices as much as possible before we investigate the totality of line-intersections (next subsection). Figure 7.5 shows the result of line-extraction on the edge-data in Figure 7.1.

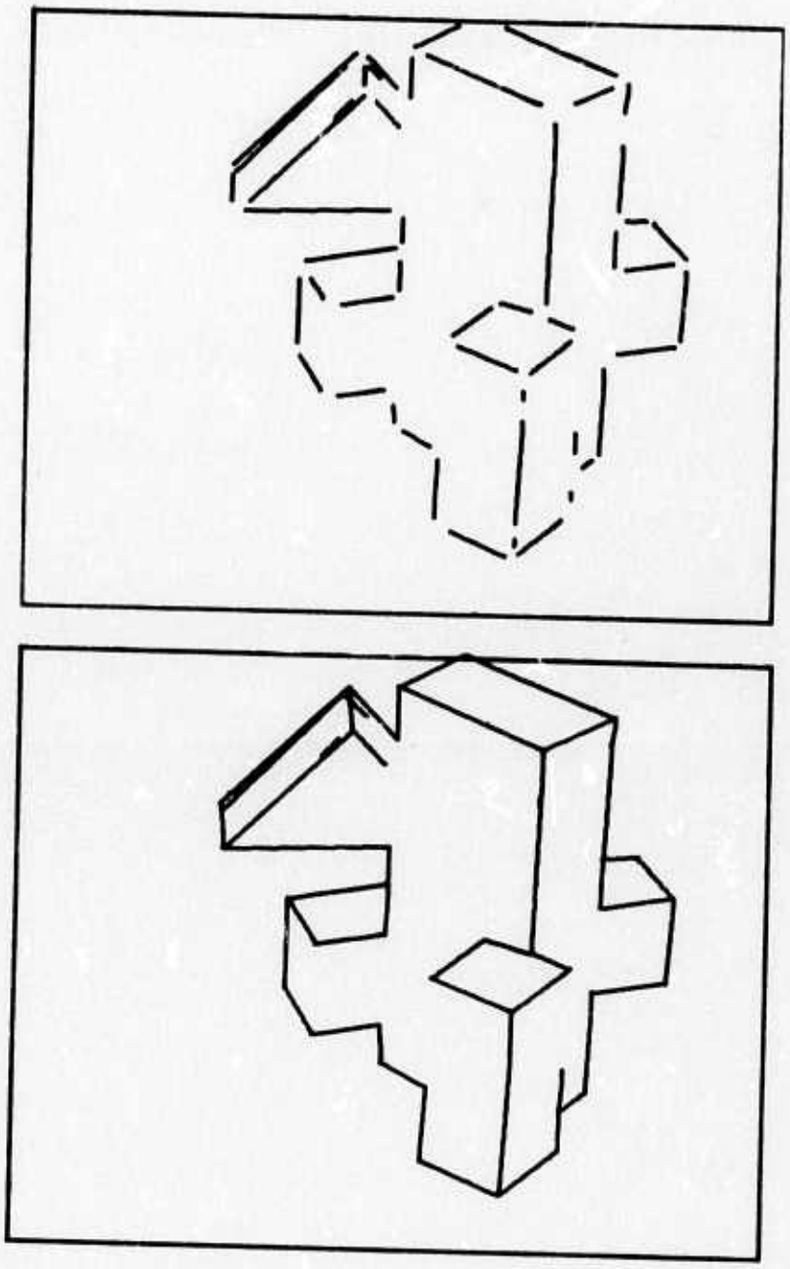


Figure 7.5

Initial lines - Tentative vertices

This concludes the preprocessing of the scene. The story continues in the section on parsing, which conveniently follows next.

8.0 THE PARSING PROCESS

8.1 PARSING STRATEGY

The word "parse" has been chosen because it describes very well what happens in this process. The parser works iteratively, extracting one object at a time, each time modifying the scene by removing the lines or segments belonging to that object. "Object" is used here and elsewhere for "object projection". The input to the parser is the initial line-drawing, which was described in the previous section.

The diagram in Figure 8.1 shows the flow of the parsing process. The first two blocks, A and B, may be characterized as preprocessing stages for each iteration within the parser. They are described in the two following subsections.

The result of the actions in block A (described in Subsection 8.2) is a pointer structure which, although the original line-drawing is unchanged, gives the tentative vertices based on intersection relations. In Figure 7.5 (bottom) that pointer structure has been used to show the tentative linkage of the lines. Note that what is shown in the figure are the connectivity relationships, using weighted vertex coordinates. The data-structure is described in the appendix, Subsection 14.1.

The tentative topology is the basis for the next step, feature

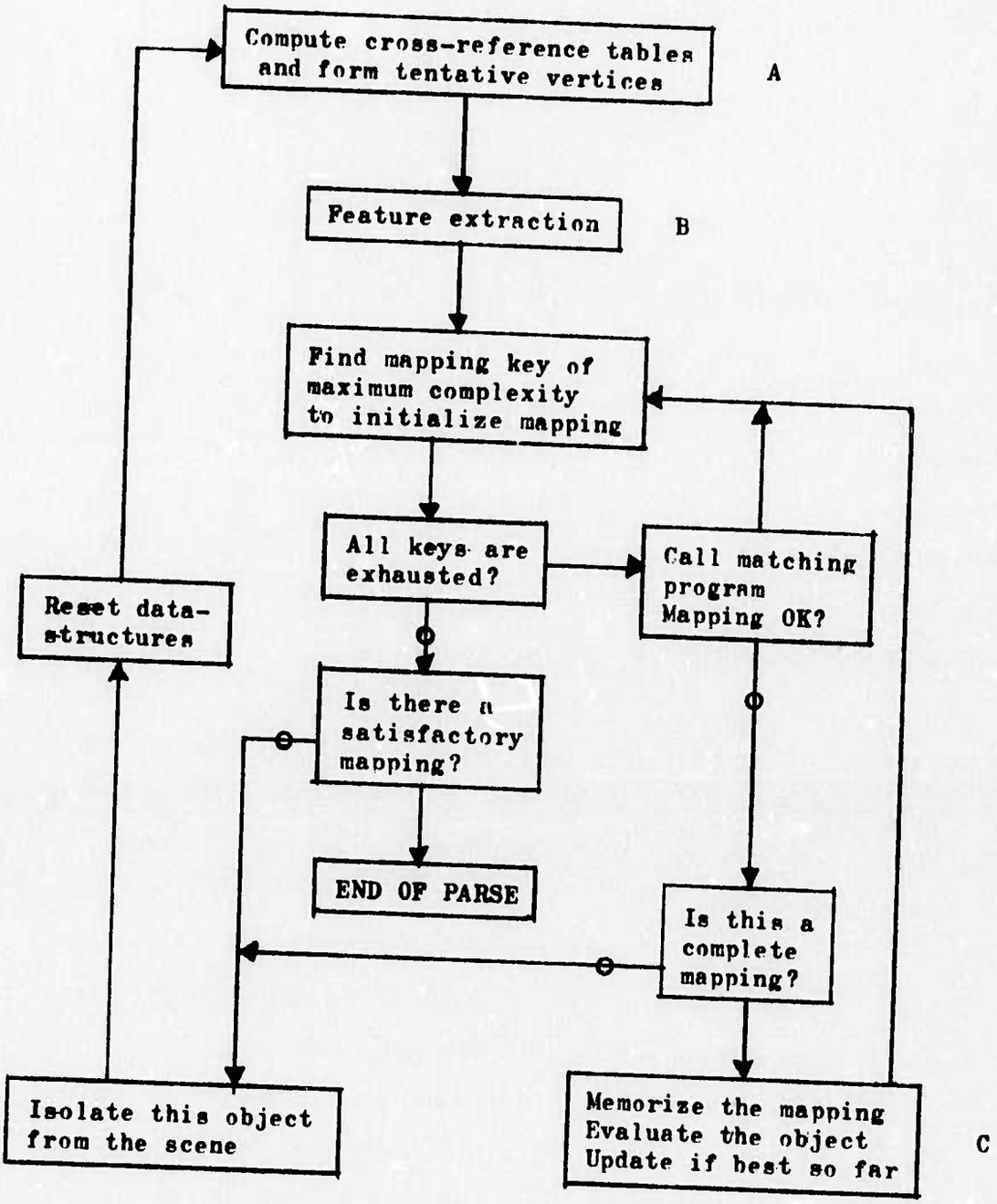


Figure 8.1

Parsing strategy

extraction. During that process, links are created between scene-elements and prototype-elements. Those links, called mapping keys, are investigated by the parser (one by one) in order of decreasing complexity until the mapping program finds a complete object or the links are exhausted. In the former case the object is accepted directly, otherwise the different mappings are compared, and the best one is chosen. Complexity of a feature is measured in terms of the number of lines involved.

The mapping routine tries to find as good a match as possible, given an initial scene-element and the prototype element it is currently assumed to map into. On return, that program has stored the best match (for that key) in compacted form for the parser to study.

The parser now compares it to the best mapping it has found so far, updating the "best"-pointer if the new mapping is better, otherwise just stepping the map-storage pointer. Thus all mappings are remembered at each iteration (but not between iterations), and before investigating a new key it is easy to check whether that key has already (implicitly) been used, i.e. if that line in the scene has already been tried for the current equivalence class and prototype combination.

Subsection 8.4 discusses object evaluation (and isolation). We must be able to decide whether one partial mapping is better than another, in order to isolate the best object. As the diagram shows (Figure 8.1), object isolation takes place when all keys have been investigated, or a complete object has been found. Since an isolated object disappears

from the current scene, the topology may subsequently have changed in some drastic way, and that necessitates a reiteration of the parser preprocessing routines. But before that we reset all data-structures to the initial line-drawing state.

So we may note that as far as the parser knows, each iteration deals with a completely new scene. The program does not remember what it did before, nor does it use its stored knowledge of previously extracted objects. It does not worry about occlusions. A match may take place even if it means that the object will partly cross over other elements of the current scene.

Such information could be utilized to some extent even in the present system, but would be fully effective only in the context of a complete, three-dimensionally based vision system.

Figure 8.2, Figure 8.3, Figure 8.4, and Figure 8.5 show the results of the iterative parsing process on our sample scene.
Figure 8.6 gives the collective final scene with no elimination of hidden lines.

There could be one more process in the total scheme, namely object completion, the idea of which would be to try combining (in turn) each of the isolated objects with the final residual line-drawing, using the matching program, in order to determine whether some partial object may be completed or at least extended. Section 10 is devoted entirely to that subject.

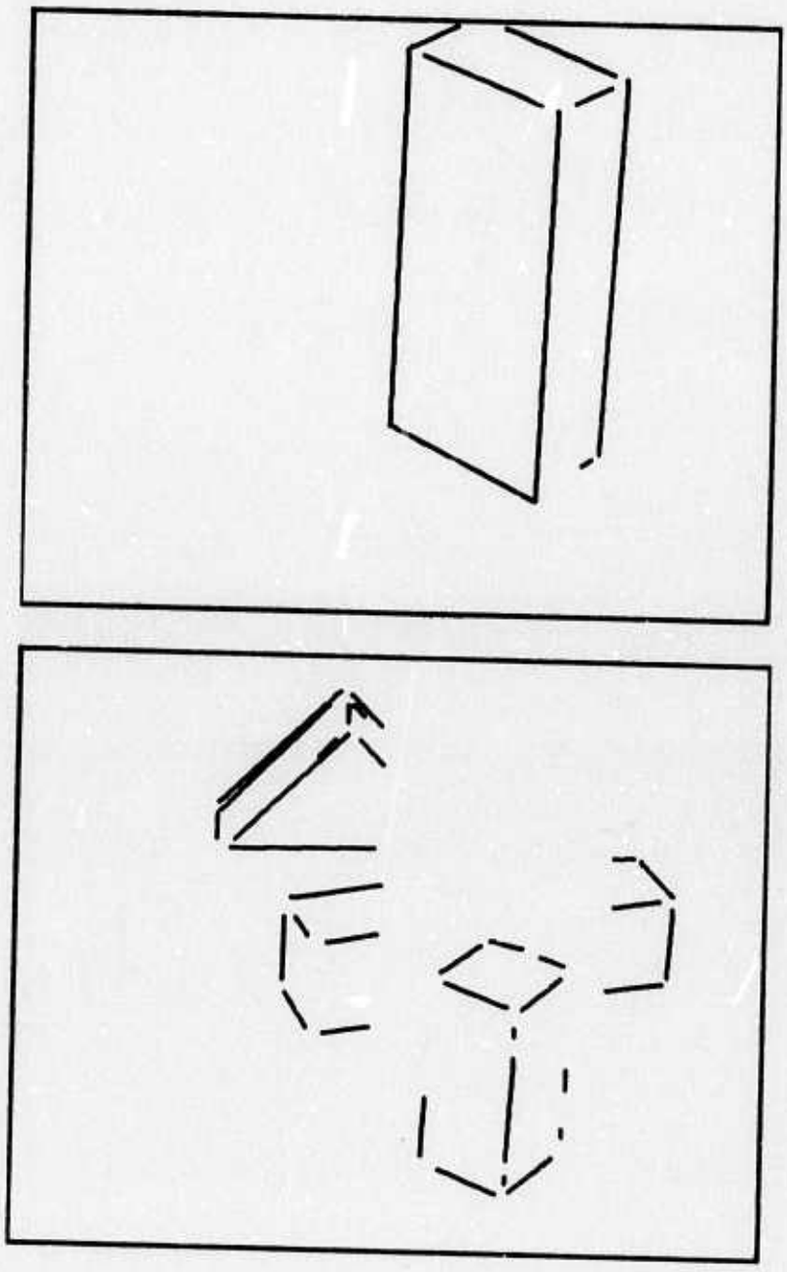


Figure 8.2

Object 1 and amended scene

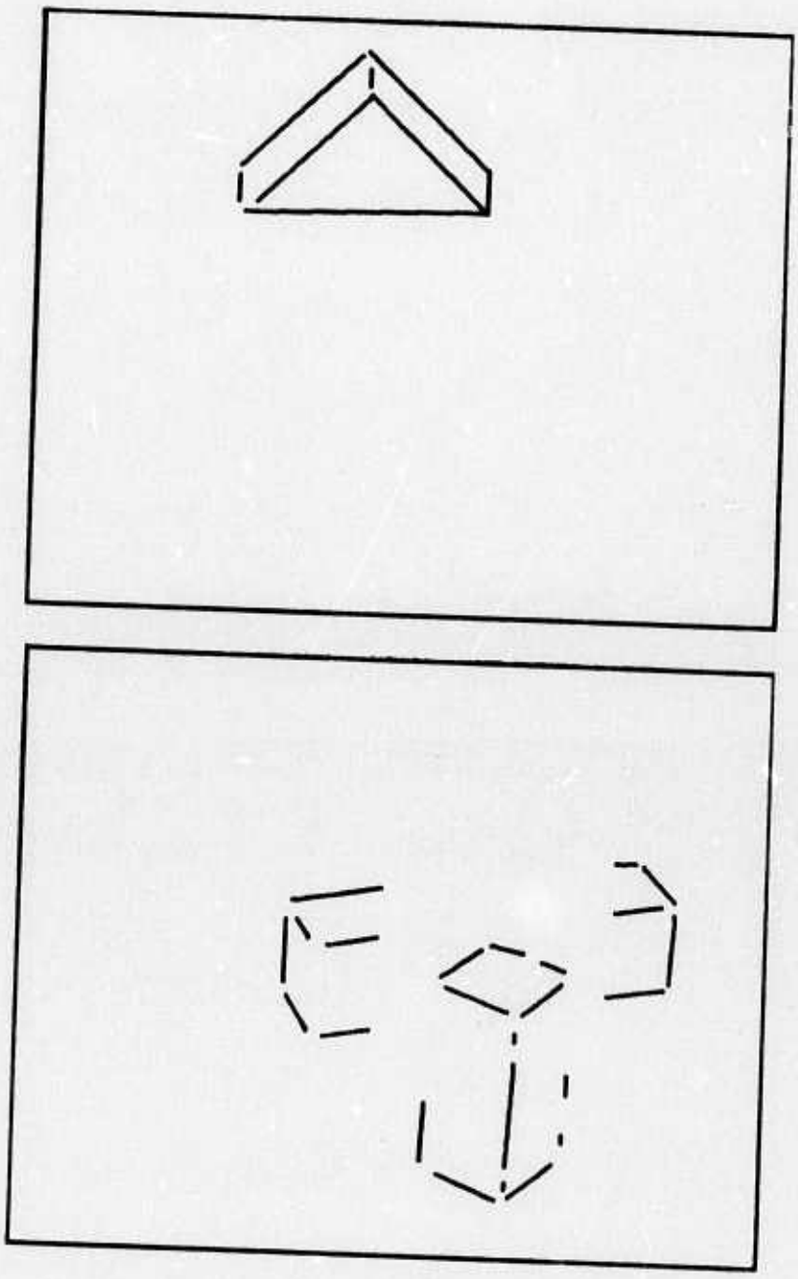


Figure 8.3

Object 2 and amended scene

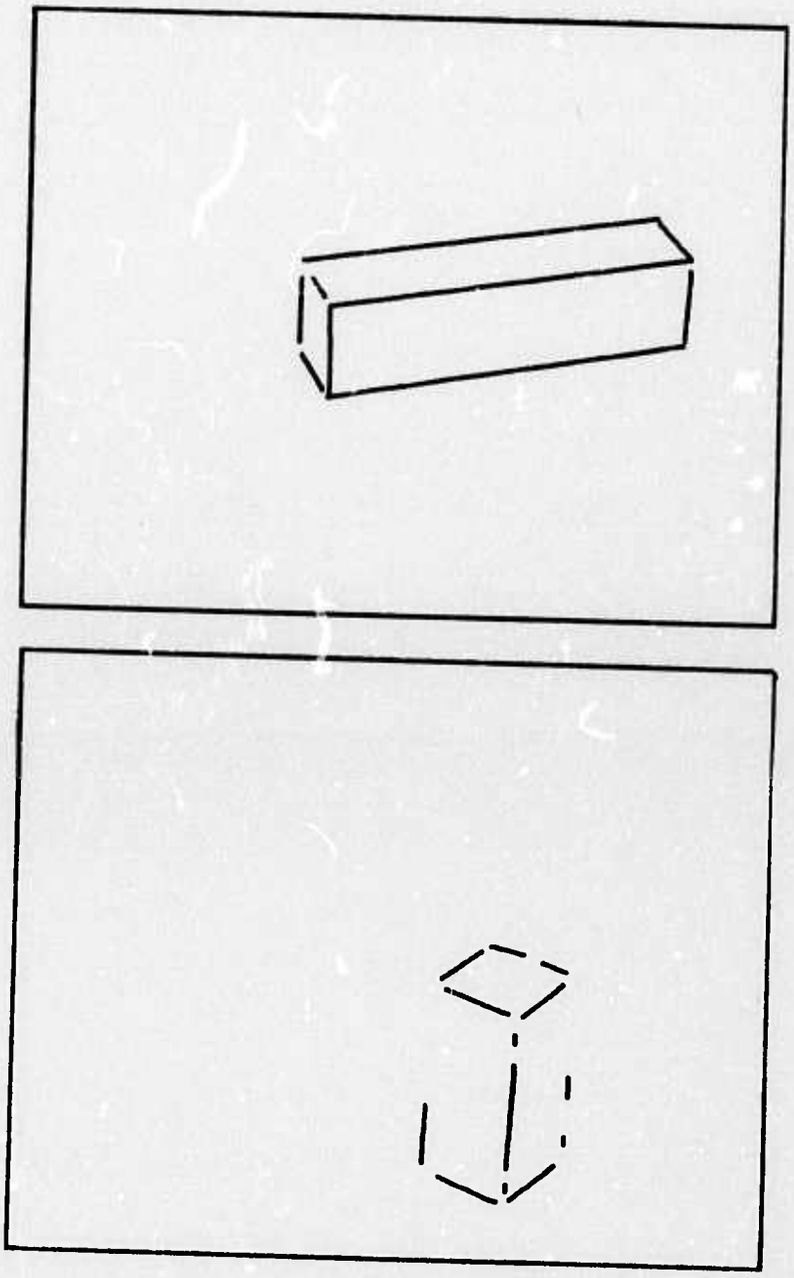


Figure 8.4

Object 3 and amended scene

8.1

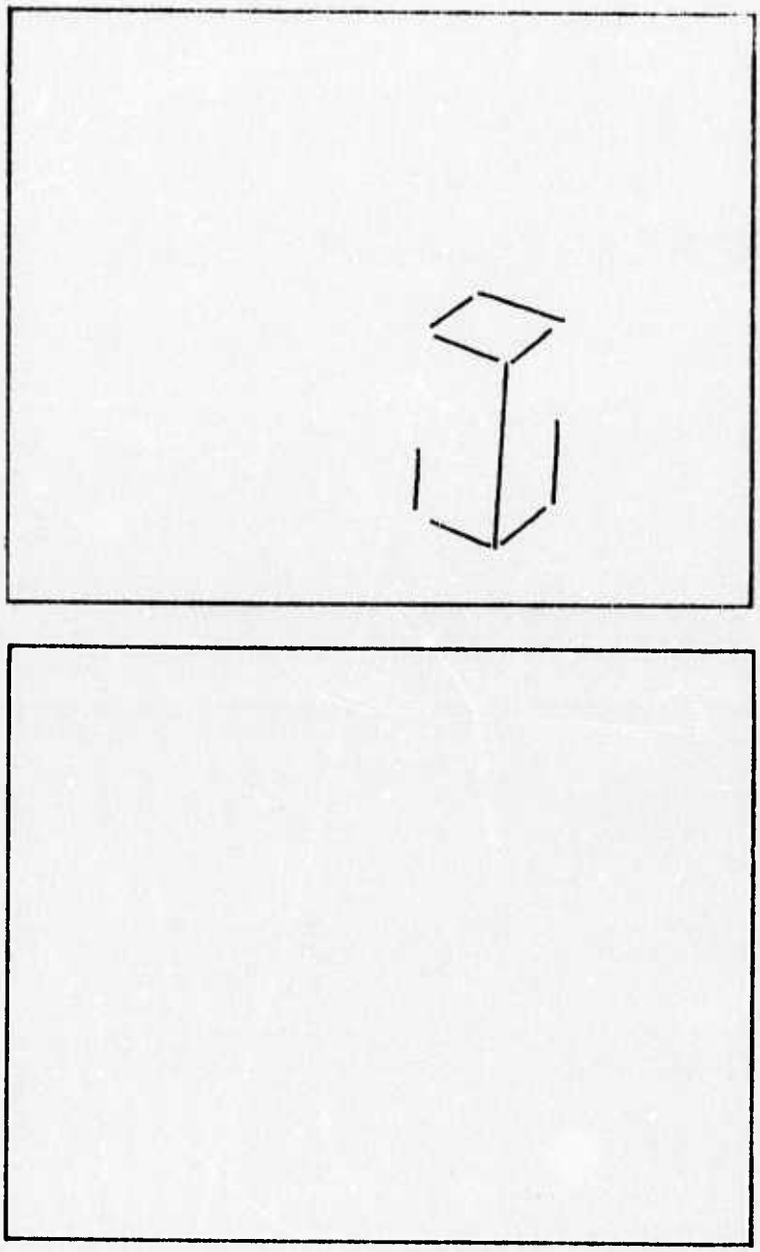


Figure 8.5

Object 4 and amended scene

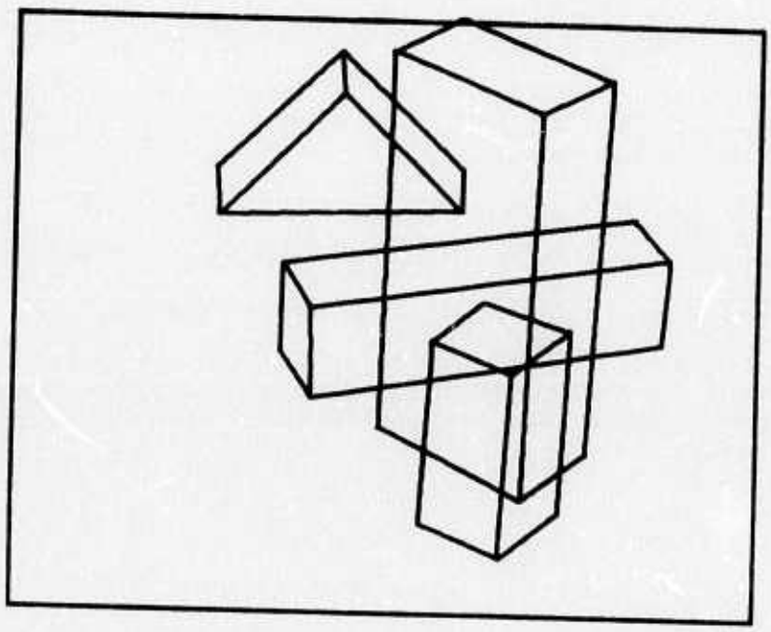


Figure 8.6

Objects superimposed

Now we proceed with three subsections dealing with blocks A, B, and C in the strategy diagram, Figure 8.1.

8.2 FORMATION OF TENTATIVE VERTICES

Since the entire parse depends on the initial mappings, and features are based on end-vertex ray-constellations for the lines, we have to somehow obtain a tentative topology in terms of linking lines together in possible vertices. This should be done fairly conservatively in order to avoid grouping excessive numbers of lines together, which would complicate the task of matching, besides possibly destroying recognizable features. On the other hand we do not want to be too conservative, either, for similar reasons.

Thus the formation of tentative vertices, with no global knowledge, is of a great deal of importance. The block diagram for this process is shown in Figure 8.7.

A "cut stop" (point B3 in that figure) is exemplified in part (b) of Figure 8.8, and consists of one line (extended) running into another. If one of the cut-off ends is short enough, a vertex could be formed here, by assuming that the short piece may be ignored.

Block A, the formation of cross-reference tables, is the process of mapping the relationships between the lines in terms of intersections

A: Form intersection cross-reference tables

B:
1. Join acceptable extension-intersections (distances OK, no obstructing lines) using restrictive parameter settings
2. Same as 1, except use full parameters
3. Join line-ends with small cut stops if and only if either end is free, giving preference to the line with the least extension, if both are eligible
4. Same as 3, except no preference
5. Extend still free line-ends into closest vertices, subject to various distance criteria
6. Join closest free line-end pairs, using liberal parameters for one of the lines

C: Iteratively, merge pairs of vertices, provided the distance between them is short enough

Figure 8.7

Tentative vertex formation

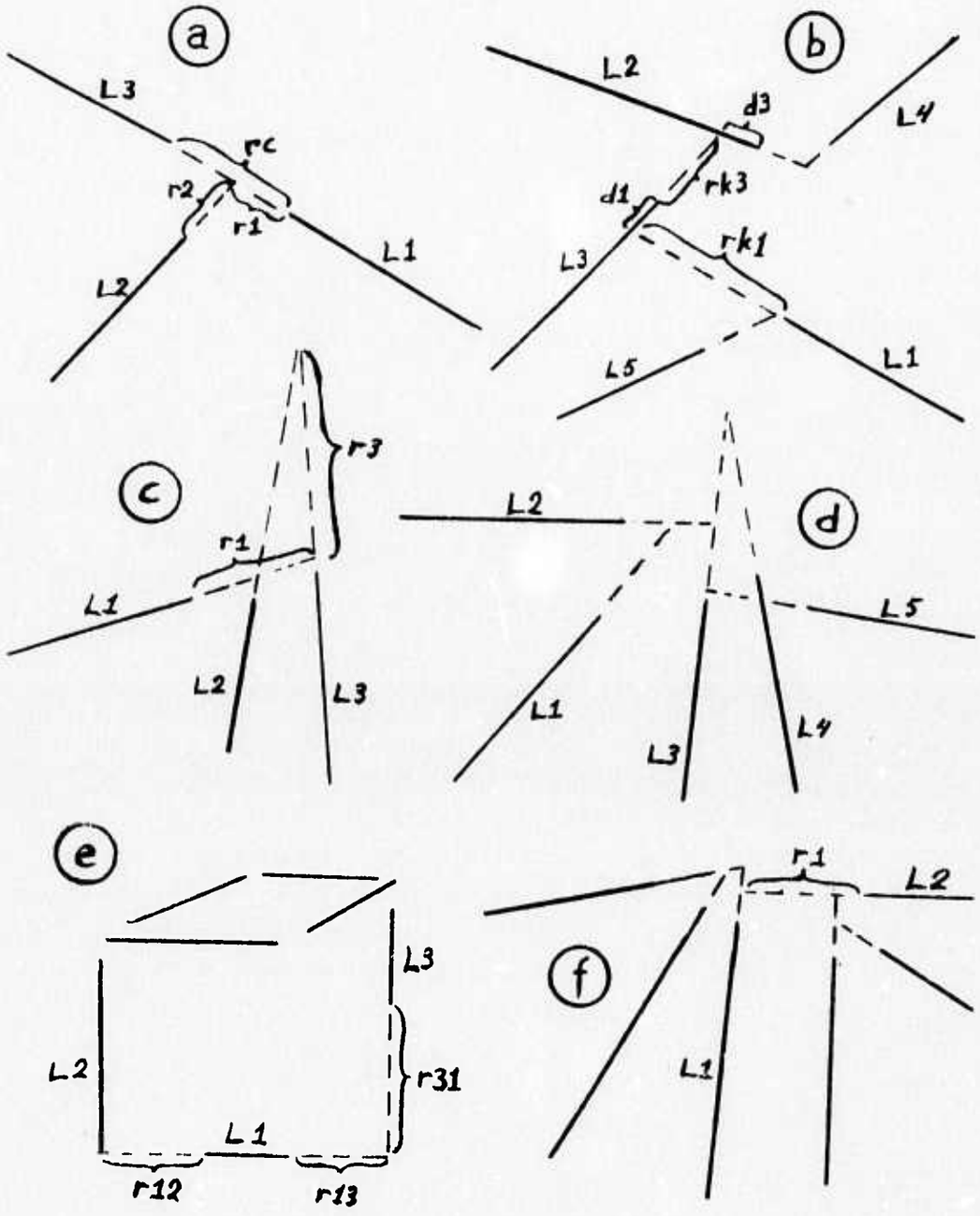


Figure 8.8

Tentative vertices - case analysis

and collinearities. This is done for every pair of lines. For each end of each line the following information results (refer also to Figure 8.8, where L1 is assumed to be the current line in all cases):

X1. Closest extension-intersection, and both distances, subject to acceptability (L2, r1, r2 in part (a) of Figure 8.8).

X2. Closest collinear line, and distance (L3, rc in part (a) of the same figure).

X3. Closest stopping line, and distances (L3, rk1, d1 in part (b)).

That block (A) is iterated once more, in order to find next-best intersections in cases where the best ones were subsequently blocked, as illustrated in Figure 8.8, part (d). Line L3 is first associated with L5, but later L4 is found to block that intersection, so that L3 is grouped with L2 instead, during the course of the second iteration.

One might of course store several best intersections for each line, to begin with. My previous preprocessor did just that (Subsection 3.2). It is basically a question of space versus time. The present scheme was chosen because (1) subsequent blockings are not overly frequent, and (2) "the best few" may be blocked as well, so that the extra code is still necessary.

Once the cross-reference tables exist, vertex formation proceeds in two steps, namely temporary vertex linking, and vertex merge. The temporary vertices are created in an iterative process where each step is conservative in relation to the previous, but where the end-result is as liberal as possible without creating confusion. I shall briefly indicate the reasons for each of the six passes, using the examples in Figure 8.8.

Pass 1 and pass 2 do the same thing, with a different extension tolerance level. Part (f) in Figure 8.8 demonstrates why. If r1 is an acceptable extension for pass 2, but not for pass 1, and if the maximum vertex-merging distance is less than r1, the two vertices in (f) are kept separate, as seems reasonable. That would not have been the case with only one pass here, since L1 would link to L2.

Pass 3 joins L3 with L2 in (b), provided the cut, d3, is small enough, and r13 is short enough. If the preference clause had not existed, L1 would have been joined with L3, and the resulting vertices would have looked differently.

Pass 4 would have joined L1 with L3, if the former hadn't already found L5, and the latter L2 (still in part (b)).

Pass 5 will permit L3 to join the others (L1 and L2) in (c), which it couldn't otherwise have done, assuming r1 and r3 are too great.

Pass 6, finally, allows L1 to join L2 (part (e)), provided r12 is in the right length-bracket, but will generally not allow L1 to join L3.

The vertex merge fuses close enough vertices, subject to some connectivity constraints. A weighted-least-square method (that takes account of line-lengths) is used in computing the best vertex coordinates for junctions of several lines.

"There are many ways to peel the banana ...". Which is the right one?

Fortunately the matching program is clever enough to be able to handle the consequences of most of the unavoidable mistakes of the ignorant procedure above.

Finally, let me once more stress the fact that the formation of tentative vertices (etc) is only reflected in the connectivity, i.e. confined to the pointer structure (Subsection 14.1), and that the initial line-drawing is in no sense affected by this.

8.3 FEATURE EXTRACTION

Assuming the program in the previous subsection has done a fair job, we should now be able to establish some links between the scene and the prototypes, in other words, recognize certain constellations as things we have seen before.

The feature extracting procedure is absolutely straight-forward.

The line-features are extracted and compared with the centrally stored ones, in a binary search through that ordered list. If a similarity is found, we store that identifier (and direction-flag) with the data for that line.

The compound features are compared to central storage in similar fashion, but as a match is found, we update that pointer structure so that the particular CF, besides pointers to various prototypes, will now also have a pointer to this specific instance in the scene. The central feature reference structure was shown in Figure 6.4.

A global switch enables the following extension in the feature extraction phase (for messy scenes). Utilizing the concept of partial similarity of line-features, an unrecognizable feature may be listed as a potential key, provided it can be reconciled to some centrally stored feature by the association only, or disassociation only, of one or more rays. Thus both adding and deleting rays simultaneously is forbidden, since such keys would be far fetched. We also want to keep changes as simple as possible.

For each unrecognized feature the central list is checked, in order of decreasing complexity, until a partial similarity is found (that conforms to the rule above), or the list is exhausted. Thus the recorded partial key refers to the maximum complexity feature which is partially similar (and reconcilable) to the one causing trouble.

This scheme may seem arbitrary (and so it is!), but we really are only

trying to provide educated, locally based guesses at this point. There is certainly an element of randomness inherent in any such local scheme. A more globally oriented, context conscious scheme would be nice, but a bit harder to design. The concept of the super-feature, discussed in Section 12 (future work), might possibly come in useful in this respect.

There are almost always enough keys to initiate the mapping process, since each iteration in the parser simplifies the scene and the topology. Again, the concept of partially similar features is utilized in the mapping heuristic, Section 9.

8.4 OBJECT EVALUATION AND ISOLATION

Object evaluation, or rather mapping evaluation, is the procedure whereby we assess the goodness of a mapping. We need to do this in order to be able to choose between different partial mappings to obtain the best.

This is one of the processes that would fare exceedingly well from being provided access to all the good things inherent in 3D consciousness (depth, occlusion, ...).

The primary evaluation is based on the following points (number of lines or rays subject to absolute as well as relative tests):

ME1. Completely mapped lines (both ends).

ME2. Incompletely mapped lines (rays).

ME3. Complete lines present in the scene.

ME4. Rays present in the scene.

ME5. Partially used (cut off) scene-lines.

ME6. Inserted lines and rays.

ME7. LF-testable lines.

ME8. Lines passing through vertices.

It should be fairly obvious in which directions (positive or negative) those items contribute in the evaluation. We want as many complete elements as possible, and we prefer that they really exist in the scene. Partially used scene-lines are abhorred, since they may be indicative of an object cut off to fit the mapping. An object (complete) is never accepted "on faith" if it contains such lines. We shall see examples of these and other exotic things later.

Finally, not all complete lines are LF-testable, that is, some of them may contain so called "assumed" rays at their end-vertices. Those are rays for which no direction can be pinpointed - only their existence is known to be a fact.

Besides the primary evaluation points, we use preference relations to decide between equally well mapped objects. The preference (somewhat arbitrarily) calls for parallelepipeds rather than wedges, non-degenerate rather than degenerate views, for instance. This is not crucially important, since the particulars of every mapping are remembered, and a post-evaluation phase could be constructed, where questionable matches could be further investigated.

Furthermore, as I have been pointing out, the system presented here is not intended as a complete vision system. Its possible role in such a system is discussed in Section 12 (future work).

OBJECT ISOLATION is simply the removal, from the active scene, of all lines belonging to the current object. The general data-structure allows parts of the scene to become part of the "subconscious" (see appendix, Subsection 14.1), and information regarding each isolated object is grouped into three different areas of the subconscious, namely:

SL1. Final mapping, existing lines.

SL2. Final mapping, inserted lines.

SL3. Line-segments belonging to the object, but superceded by inserted lines.

We will get back to this subject in connection with the discussion of a possible object completion phase (Section 10).

Some of the items in this subsection may seem slightly undefined at this stage, but everything will become clear as we now proceed to the tale of the matching process.

9.0 PROTOTYPE MATCHING

9.1 STRATEGY OUTLINE

The process of (partially) mapping a prototype onto elements of the scene is crucial to this vision system. That task is by no means trivial - the task of documenting it isn't either. The process is a fairly complex recursive one, which I will do my best to describe clearly and concisely, using hierarchies of diagrams with pertinent comments in the text.

The general idea is as follows:
Assume we have an initial correspondence between one directed line in the scene and one directed line in a prototype (I use "line" here, even though it is not endowed with coordinates).

We now try to establish correspondences between the rays emanating from the ends of those two lines. If we are lucky, those scene-rays are all in their proper places. However, realistically, they very often are not. There may be too few, or too many, or (even if the numbers agree) they may not be pointing in the right directions. Different alternatives must then be investigated, and the line-features are used as prediction guides in this context.

Another problem is the fact that lines may have been broken up,

sometimes equipped with false but plausible-looking vertices, sometimes with pieces missing. Other lines just die in the middle of nowhere, in which case we must attempt to get to them from elsewhere in the topology.

Thus the matching process is of necessity recursive, since in general we have to investigate the consequences of alternate choices, recursively, before finding an optimal mapping. As a first approximation, each mapped vertex defines a recursive level. If that mapping is successful, we then try another line-end in that extended context, bumping the level. Ideally this carries on until the prototype has been matched.

In practice we may come to a grinding halt for many reasons, a few of which are:
Two unfusable scene-lines (or two different vertices) are put into an identity relation by consequence of the mapping.
A line-feature does not check.
Two lines extended to a vertex intersect in a topologically impossible place.
A line is too long or too short.

Error conditions will be described in detail later, suffice it to say here that we meet with conditions that necessitate recursive back-up. Backing up to level R, we then investigate the next choice at that level. All levels accounted for, the tree may become quite large, but it is kept down to size by the use of features, as well as parallelity and length classes, for screening purposes. We shall see this later.

Thus, given the initial mapping between two lines, the idea is to work down the topology of the prototype, matching those elements with scene-elements until a complete match is found or the recursive process is exhausted, in which case the result is a partially mapped object. In that process there is a maximizing mechanism, ensuring that we exit with the best such partial match.

We do not try to work from several different vantage-points at once, e.g. trying to link up several individually recognized features or regions into a partial or complete object, although that might be a possibility to be used in conjunction with the present scheme. Especially with good initial line-drawings.

The recursion has been programmed explicitly (as opposed to using recursive procedures), for several reasons. We only have to save a very limited amount of information at each level. We should like to be able to have access to all stages of recursion at once, and to be able to easily back up to any desired level. We also save time and space.

The first diagram, Figure 9.1, illustrates what has been said above. It is simplified in the extreme, and in this section we shall proceed to clarify the specifics of the process, giving diagrams for each building block by itself (as far as possible), such as back-up, line-merge, ray-identification ...

First, however, we shall describe the mapping data-structures and deal with the concept of partially similar line-features, which is used for purposes of hypothesis formation regarding new vertices.

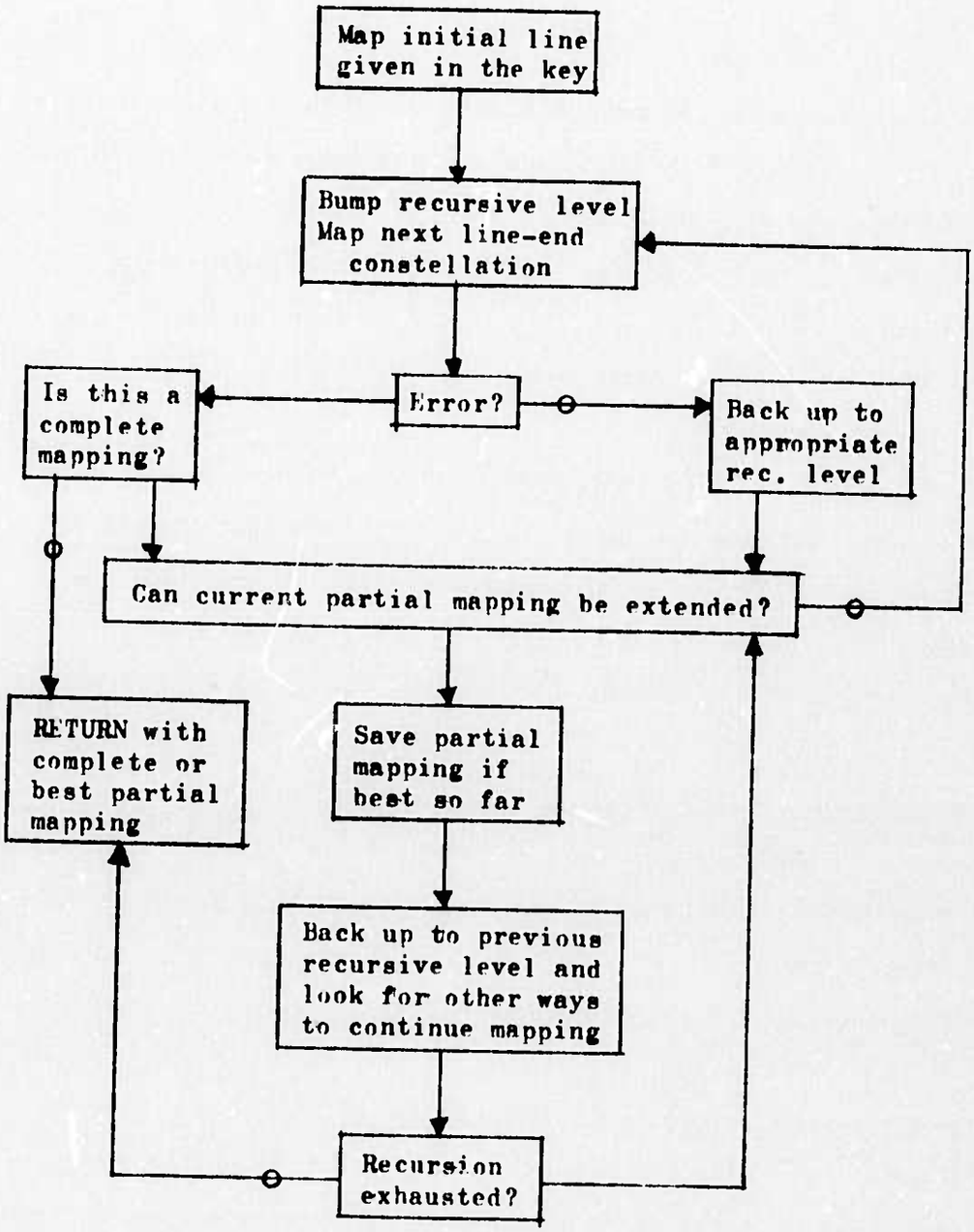


Figure 9.1

Simplified matching strategy

9.2 DATA-STRUCTURES

As a rule I do not like to burden the presentation with details, but in this case they serve the honourable purpose of clarifying the rest of the section, and making it easier to describe. For undefined terms, see Section 2.

The first structure is the template, that is the expanded prototype topology structure, see Figure 9.2. For each line, LENDV names end-vertices, LENDP names orbital successor lines, PARCLA contains the parallelity classification, and LENCAT the length categorization within that class. There is also storage for the physical entities associated with those categories.

Figure 9.2 also provides an example to illustrate this. The lines are ordered the same as their LF:s, as has already been pointed out. Since the general data-structure is organized similarly (see Subsection 14.1), it is easy to search the topologies of prototype and scene in parallel, setting up and checking correspondences.

The length-class information is used throughout the matching process, for discriminatory purposes, allowing for some tolerances, of course. In cases where the prototype indicates perspective deformations, those tolerances are more liberal, in the proper directions.

Figure 9.3 shows some of the structure used for recursion.
MAPORD is a vector containing (in the timewise order of mapping) the

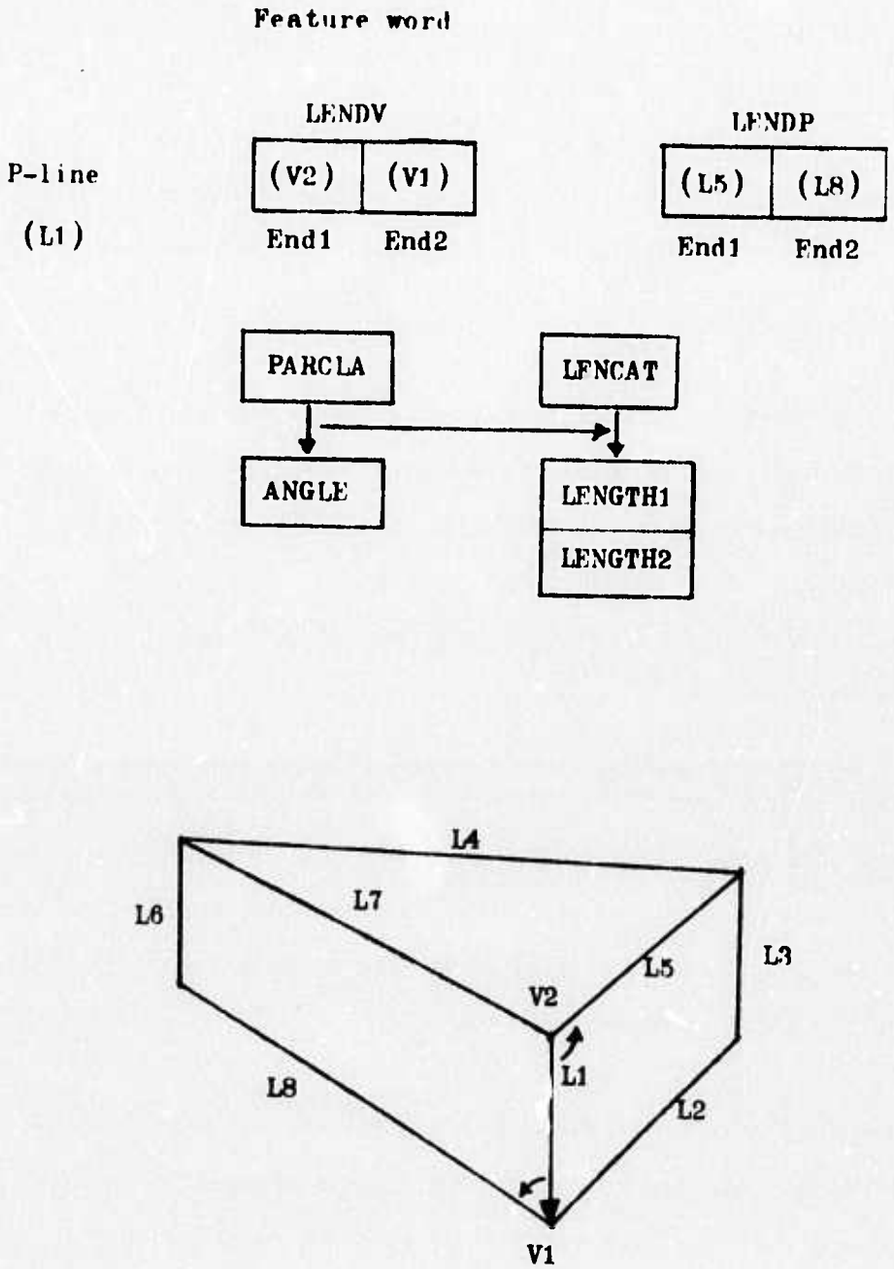
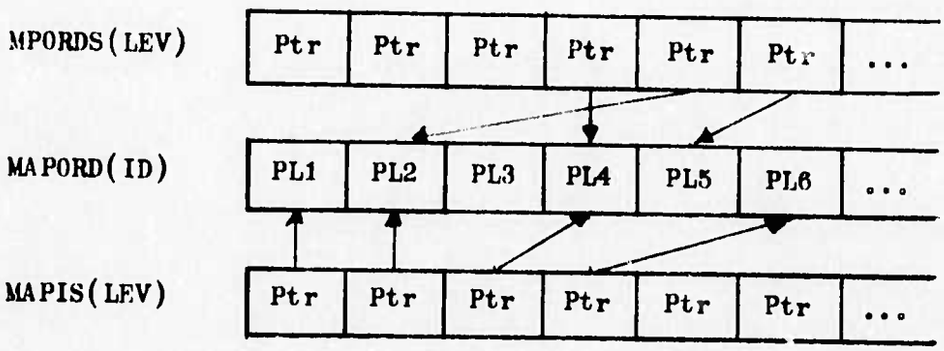


Figure 9.2

Expanded prototype structure



PLMAP(P-line,End)

Scene line-end mapped

PLMAPO(P-line,End)

Scene line-end mapped

LLEV(P-line,End)

Rec. level of mapping

LLEVO(P-line,End)

Rec. level of mapping

PVMAP(P-vertex)

Scene-vertex mapped

VLEV(P-vertex)

Rec. level of mapping

Figure 9.3

Recursion data-structure

prototype lines referenced so far. The corresponding scene-elements are stored in PLMAP etc.
MPORDS is indexed on recursive level, and contains pointers to the currently referenced MAPORD entries, at each level. As we shall see, those pointers may cross.
MAPIS, also indexed on level, contains pointers to the last MAPORD items created at the different levels.

Thus the main mapping alternatives are listed in MAPORD, but there are usually several possibilities for each one of those entries. I feel I should clarify one thing already, namely that a new MAPORD entry is created if and only if a previously unreferenced prototype line is encountered in orbiting a vertex. Furthermore, a MAPORD entry constitutes a mapping alternative if and only if that P-line is unmapped at one end, i.e. that end-vertex has not been orbited (modulo recursive back-up).

In the same diagram, Figure 9.3, we demonstrate how mappings are recorded.
PLMAP contains, for each prototype line end, the corresponding scene element.
LLEV, indexed in parallel, contains the recursive level at which that mapping took place.
PLMAPO and LLEVO are 1-level push-down stacks for PLMAP and LLEV, used when a line is being replaced (in the creation of a new vertex, or connecting two vertices), as will be explained later.
PVMAP and VLEV store vertex information.

The final item in this subsection deals with the line-fusion mechanism. Fusions deal with collinearities, and only take place for base-lines (every vertex mapping has a base-line, the MAPORD entry).

If we want to explore a new branch at some level, we check (iteratively) whether the present base-line may be extended. If that happens to be so, that alternative is tried and the fact is recorded for the P-line end, at which the fusion took place.
LFUSE is a stack (6 levels), for each P-line end, containing packed pointers (a sixpack indeed) into a common area.
LFUSES is that common area, where each word supplies enough information (about the fusion of two lines) to enable proper back-up, if and when necessary.

At a fusion, which is always based on existing collinearity links, a new compound line is created and linked into the data-structure, while the constituents are shoved into the subconscious. Note that the fusion and line replacement mechanisms use different stacks, and are therefore independent.

We shall now turn to a discussion of partially similar line-features.

9.3 PARTIALLY SIMILAR LINE-FEATURES

Since the line-feature is used in checking all lines and their end constellations, it seemed a natural thing to use it also in determining what was wrong (and what should be done by way of correction) if the check was negative.

The input to this algorithm consists of two feature words, and two direction bits. The first feature word is used as a template. The second is the one to be checked, and for which (if necessary and possible) corrections should be indicated. For reasons of sanity we require that one end of the second feature be OK (this is checked, of course). The requirement is also a realistic one as regards the way in which the model is traversed, one line-end at a time, such that the other end is mapped previously.

I shall give no details of the program here - it is straight-forward - only the format of the modification word (MODIF), which is one of the outputs, and a few examples. All of that in Figure 9.4. Orbits start from the base-line, and the rays are referenced in that order.

As "bare" we define a vertex with as many insertions as there are rays (excepting the base-line). The entire MODIF word (part (a) in the figure) is defined as "ambiguous" if and only if there is an ambiguous ray position somewhere, i.e. e.g. if we do not know which of several rays to delete.

Part (b) gives the template for cases (c), (d), and (e), in which the

(a)

Special bits (first 2)	List of action-elements (2 bits for each ray at the vertex subject to modification)

Codes		Codes	
00	Unambiguous, not bare	00	No change
01	Unambiguous bare	01	Insert ray here
10	Ambiguous, not bare	10	Delete this ray
11	Ambiguous, bare	11	Ambiguous

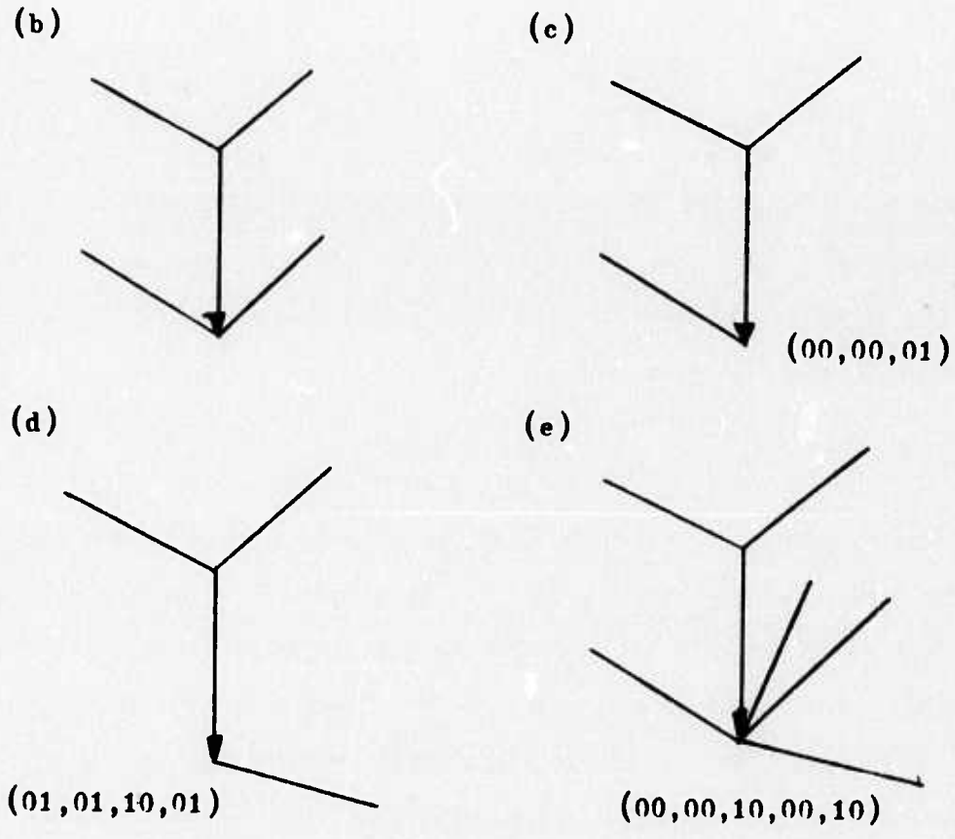


Figure 9.4

Line-feature modification word - MODIF

lower vertices are the ones to be compared and corrected. The resulting MODIF words are given in the figure as well. Of course, if there is no change, MODIF is set to zero.

The MODIF word is subsequently deployed as a template for the mapping of rays at the vertex currently being investigated, as we shall see later.

The following subsection describes an extension of the concept treated here.

9.4 LF MODIFICATION RECONCILIATION

This idea arose due to the fact that a vertex may sometimes be ambiguous from the direction of one line, but unambiguous from another. Figure 9.5 shows an example of this.

The prototype context is given in (a), and the scene in (b). The vertex under investigation is of course V, at the intersection of L1 and L2. Suppose we are dealing with the line L1, in an effort to map the rays of V. The feature template (c), and the scene-feature (d), illustrate the situation. The ray L4 is easily seen to be superfluous, but L5, L3, and L2 are all converging with L6, so the program, knowing that at least two of them have to be deleted but not knowing which two, marks all three as ambiguous. However, if L2 were the base line (f), and LP2 the template (e), then the situation is unambiguous (based on the parallelity bit).

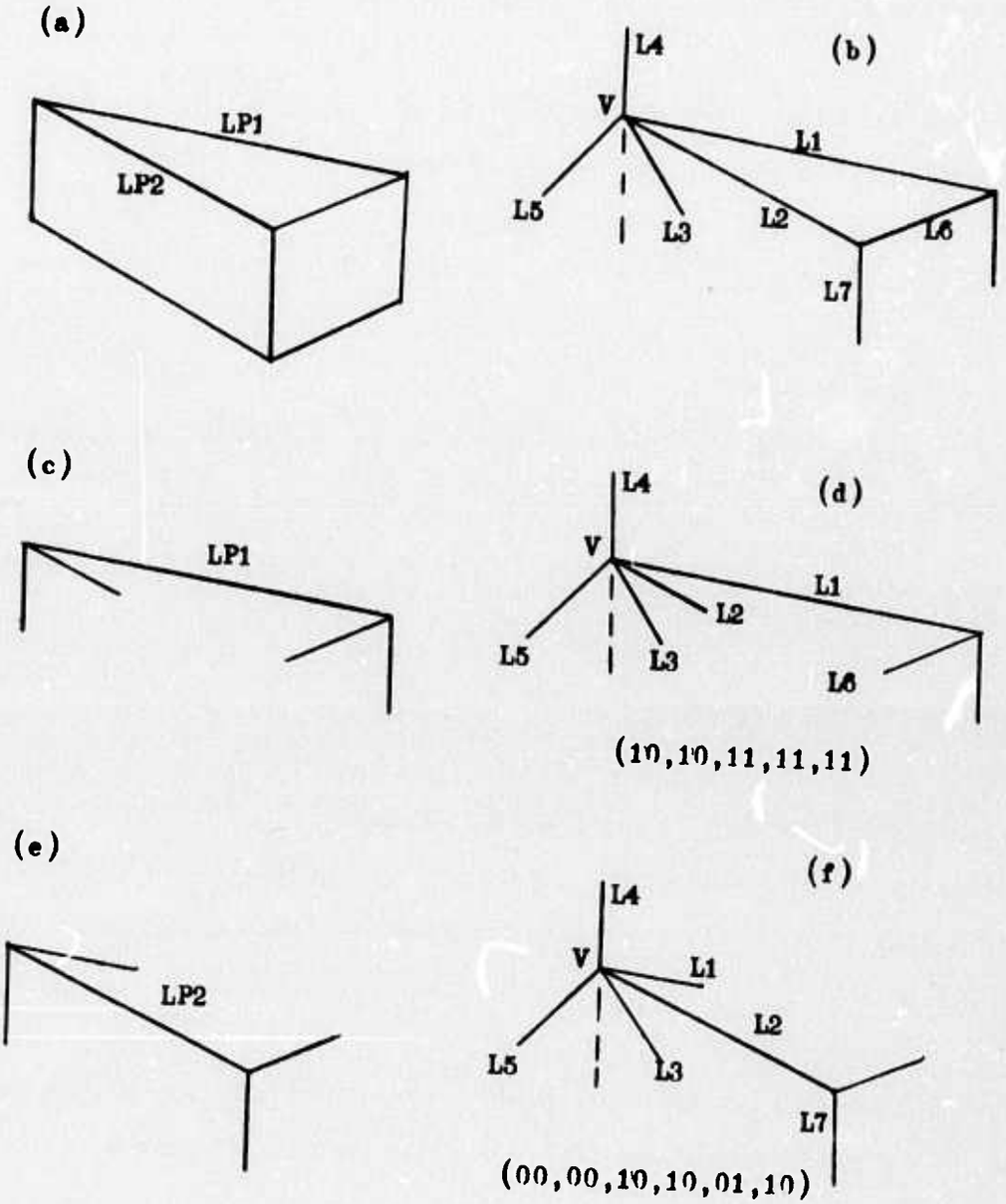


Figure 9.5

LF modification reconciliation

We introduce the following definition:

RECONCILIATION of a feature modification word from one line to another (at the same vertex) is the process of rearranging the information of that MODIF word so as to make it applicable from the vantage point of the second line.

In our case, reconciliation of MODIF from L2 to L1 is a way to disambiguate MODIF of L1, using MODIF of L2 and the connectivity of the vertex.

The following algorithm is used:

RECONCILIATION ALGORITHM.

Assume the MODIF word is [M,A1,A2,A3, ...,An,00, ...,00], where the A:s stand for 2-bit action items, and M for the two characteristic bits, in this case [00], since we are only interested in reconciling useful MODIF:s.

Let PL1 and SCL1 be the template and scene element for the MODIF word, which is to be reconciled to PL2 and SCL2. We then define the following quantities:

DP = Orbital distance from PL1 to PL2.

DSC = Orbital distance from SCL1 to SCL2.

DL = Number of action elements indicating "leave", [00], up to and including "present" element in MODIF.

DI = Same for "insert", [01].

DD = Same for "delete", [10].

We then look for that action element of MODIF, for which

$$DP = DI + DL$$

That element must be a "leave", [00], and we also must have

$$DSC = DD + DL$$

This ensures identity of the second line in prototype, scene, and MODIF word, so that the reconciliation to that line may take place. The new MODIF word will then have the following format, assuming the action element found above is Ak:

MODIF(rec) = [M,Ak+1, ...,An,Ak,A1, ...,Ak-1,00, ...,00].

In our example, Figure 9.5, MODIF for L2 is reconciled to L1 thus:

DP = DSC = DL = 1 DI = DD = 0

MODIF(old) = [00,00,10,10,01,10, ...]

MODIF(rec) = [00,10,10,01,10,00, ...]

So we get the correct indication of the need for an inserted ray parallel to L7, while L3, L4, and L5 are all branded as non-conformists and eliminated.

9.5 MORE GENERALITIES - EXAMPLE

The initial mapping references one scene-line, and it is assumed that the rays emanating from that original line all map into the corresponding prototype elements. Of course we know that the line-features agree, by definition. Thus the first three recursive levels are:

Lev1. The original line, provided by the key.

Lev2. The first end-vertex, and its rays.

Lev3. The other end-vertex, and rays.

We never allow recursive back-up to reach level 3, or below. Once established, those mappings remain fixed. The reason for this is that levels 1, 2, and 3 all refer to mappings directly involving the key, and it would not make sense to provide back-up past that stage.

We note here that the initial mapping is always given as above, regardless whether the key was provided by a line-feature or a compound feature. In the latter case, the first line referenced by the feature is used. The other line will certainly be mapped later, since those features of object and prototype are in agreement.

There is one inequity here - which I hasten to admit before being found out - and that is the fact that the second line of a CF, being mapped at level 2 or 3 at one end but 4 or more at the other, is subjected to recursive back-up (and thus to extension, for example). If we really wanted to push things we should also (when the results so indicate)

investigate the case of having the second line as the original. This would only very rarely make a difference, however, and so is not worth the extra computing.

This is especially so since, if the parsing process reaches the state of dealing with LF-mappings as well, this second alternative will be investigated (but the first one, of course, will not be repeated).

There are two basic phases in the recursive process. The first phase exhausts the LF-consistent (base-lines) mappings ("F-mappings") - does not accept any others - and thus branches out over the most dependably mapped part of the topology. The second phase deals with more difficult mappings, utilizing partial feature similarity and reconciliation. During the second phase we may get back into elements mapped during the first, due to recursive back-up, by this time they are not treated as special. The second-phase mappings are called consequence mappings ("C-mappings").

In order to make these things a bit clearer I have provided a simple and complete example of the typical actions of the mapping process. It does not contain any of the more exotic pathologies, only one partially missing line and a couple of superfluous ones. It does not necessitate recursive back-up. Figure 9.6 gives prototype (a) and scene (b), and also illustrates various stages of the mapping. The table below demonstrates the order of the mapping process for this example. Quantities in parentheses refer to the scene, others to the prototype. First and second ray refer to elements being referenced for the first

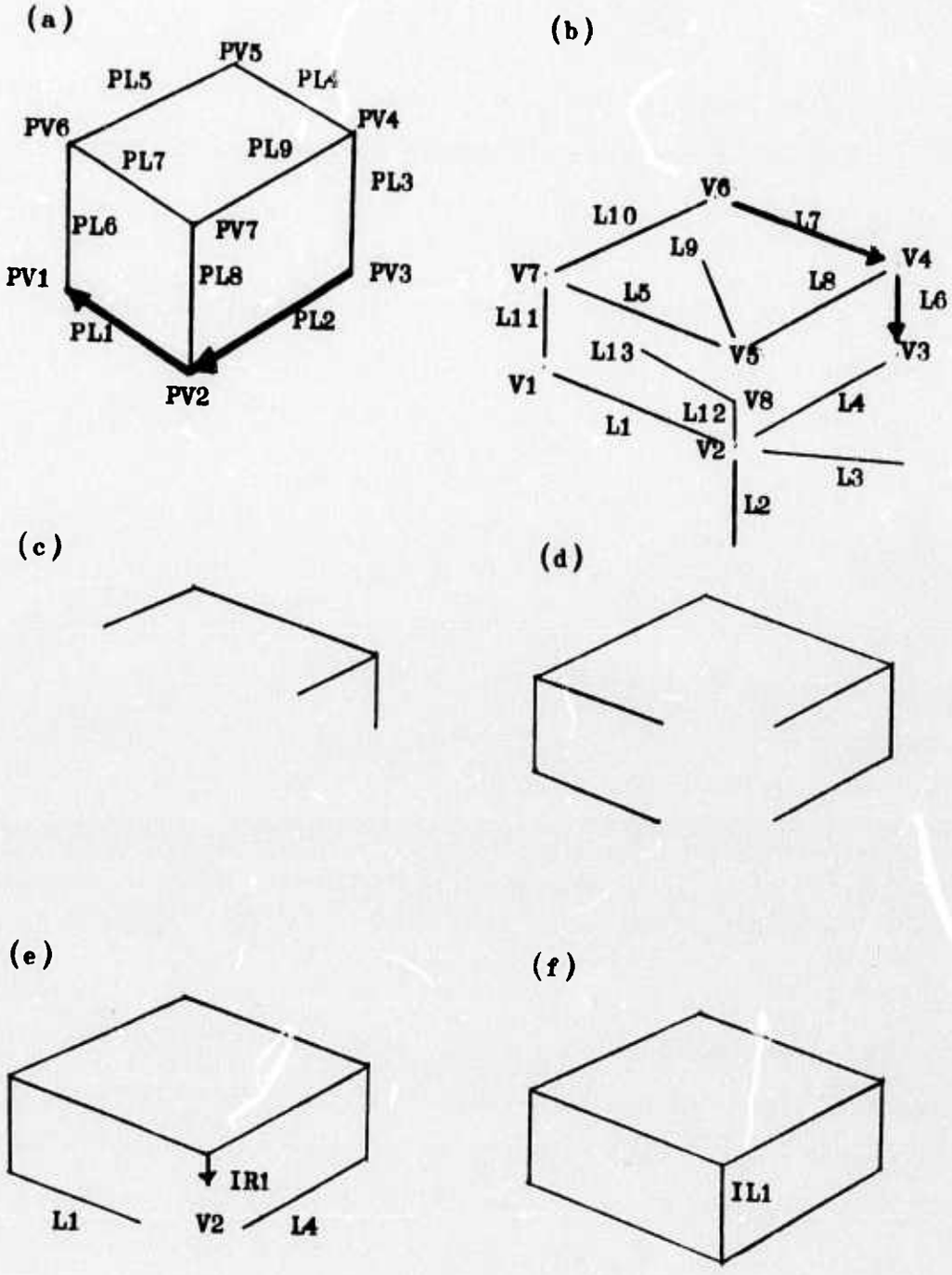


Figure 9.6

Simple example of mapping process

time. This is the order in which they are introduced into MAPORD (Figure 9.3).

Table 9.1

Order of exemplified mapping

LEVEL	ORBITED VERTEX	FIRST RAY	SECOND RAY
1		PL2 (L7)	
2	PV3 (V6)	PL3 (L10)	
3	PV2 (V4)	PL8 (L8)	PL1 (L6)
4	PV4 (V7)	PL4 (L11)	PL9 (L5)
5	PV1 (V3)	PL6 (L4)	
6	PV5 (V1)	PL5 (L1)	
7	PV7 (V5)	PL7 (IR1)	
8	PV6 (V2)		

Here are some comments:

Levels 1, 2, and 3 constitute the initial mapping, (c) in the fig. Levels 4, 5, and 6 represent additional F-mappings, almost completing the object, (d).

At level 7 (part (e)), partial similarity (V5 of L8 and PV7 of PL8) was used, discarding L9 and inserting the tentative ray IR1 (of unit length), based on the parallelity class of L6 and L11. IR1 is then linked (one way) with L12, since they are found to be collinear.

At level 8 (part (f)), finally, partial similarity for L4 and PL6 is

used to get rid of L2 and L3. Finding that the other end of PL7 is mapped into IR1, and using the collinearity, we decide that the mapping of PL7 is OK. We insert the compound line IL1, and find that the object is now complete. All lines are LF-testable and OK.

Note that if partial similarity hadn't been able to make sense of the situation at V2, that vertex would have been constructed basically as an intersection of L1 and L4. Then IL1 would have been inserted as a replacement for IR1.

The next subsection gives a fairly detailed account of the recursive process.

9.6 THE RECURSIVE PROCESS

The presentation will center around six diagrams. The first provides the main flow. The second deals with vertex orbiting and the third with ray mapping. The next two explain the functions of erasure and back-up. The last diagram gives a detailed account of the routine taking care of back-up. Messy though some of them may seem, the flowcharts only record the main actions or branches. Minute detail is of course unnecessary for the purposes of this presentation.

Figure 9.7 is the main flow diagram. Simply stated, we study "the next available alternative", that is, the next un-orbited P-line end in the

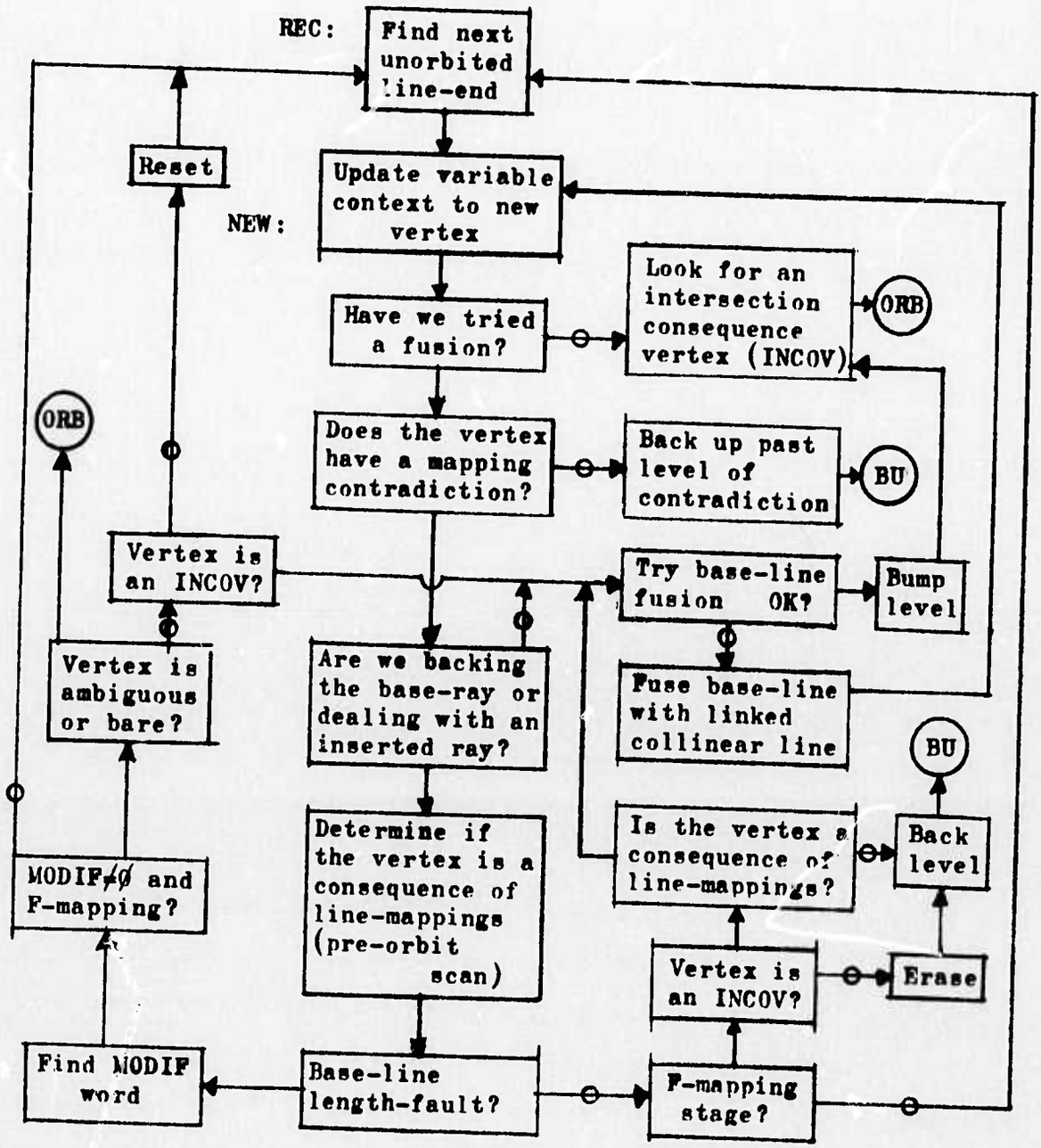


Figure 9.7

Main flow of the mapping process

MAPORD-list. This end may already be flagged as backing up, which means that it is either an inserted ray or that we have investigated it before, and are now left with the final alternative, namely regarding the line as a ray. In that case we see if there is enough information in the prototype topology to determine an intersection consequence vertex ("INCOV"). Thus, in part (e) of Figure 9.6, the vertex V2 would be an INCOV of L1 and L4, since their prototype counterparts, PL4 and PL5, are linked at PV5 (part (a)).

Otherwise we study the vertex, using the MODIF word (Figure 9.4) to decide what actions to take. If the vertex is ambiguous or the base-line would be a ray before insertions, we look for an extension of the line. The diagram should explain most of this. The "pre-orbit scan" simply finds out whether the vertex is mapped by consequence of two lines, as above, which influences the branching.

The diagrams in Figure 9.8 and Figure 9.9 demonstrate the process of mapping a vertex-constellation (orbiting a vertex). The diagram should be more or less self-explanatory. The heart of it is the referencing of one ray-position at a time, and the MODIF-based action decision at that point. When the vertex has been orbited we check the bareness again (a ray may have been replaced on the basis of its angular argument), and we demand that the finalized vertex contain at least one scene-ray besides the base-line. If that is the case, we then check that all new two-way mapped lines are LF-consistent.

Figure 9.10 explains the labels ERASE and BU in the previous diagrams,

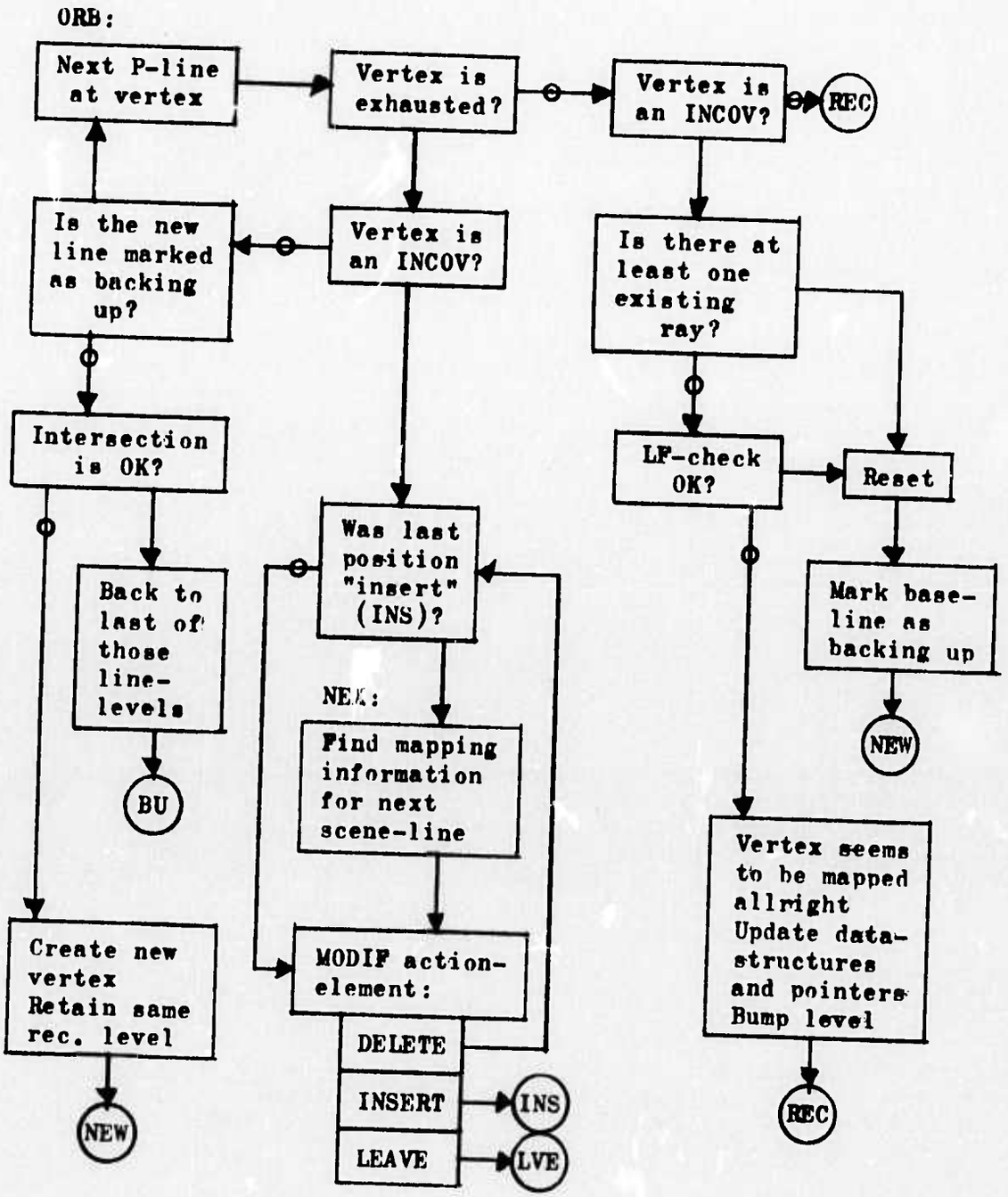


Figure 9.8

Vertex orbiting - mapping - process

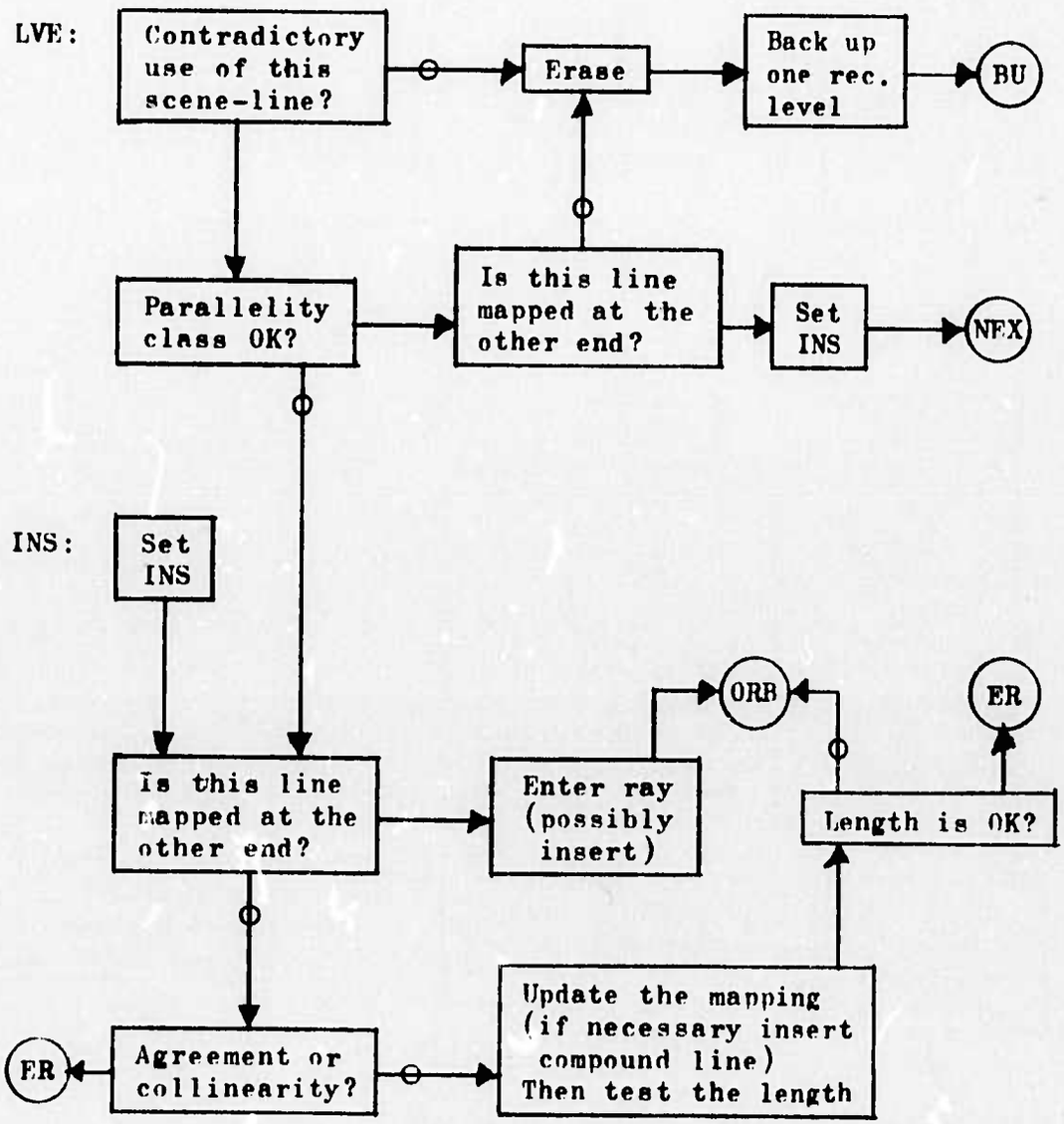


Figure 9.9

Ray mapping

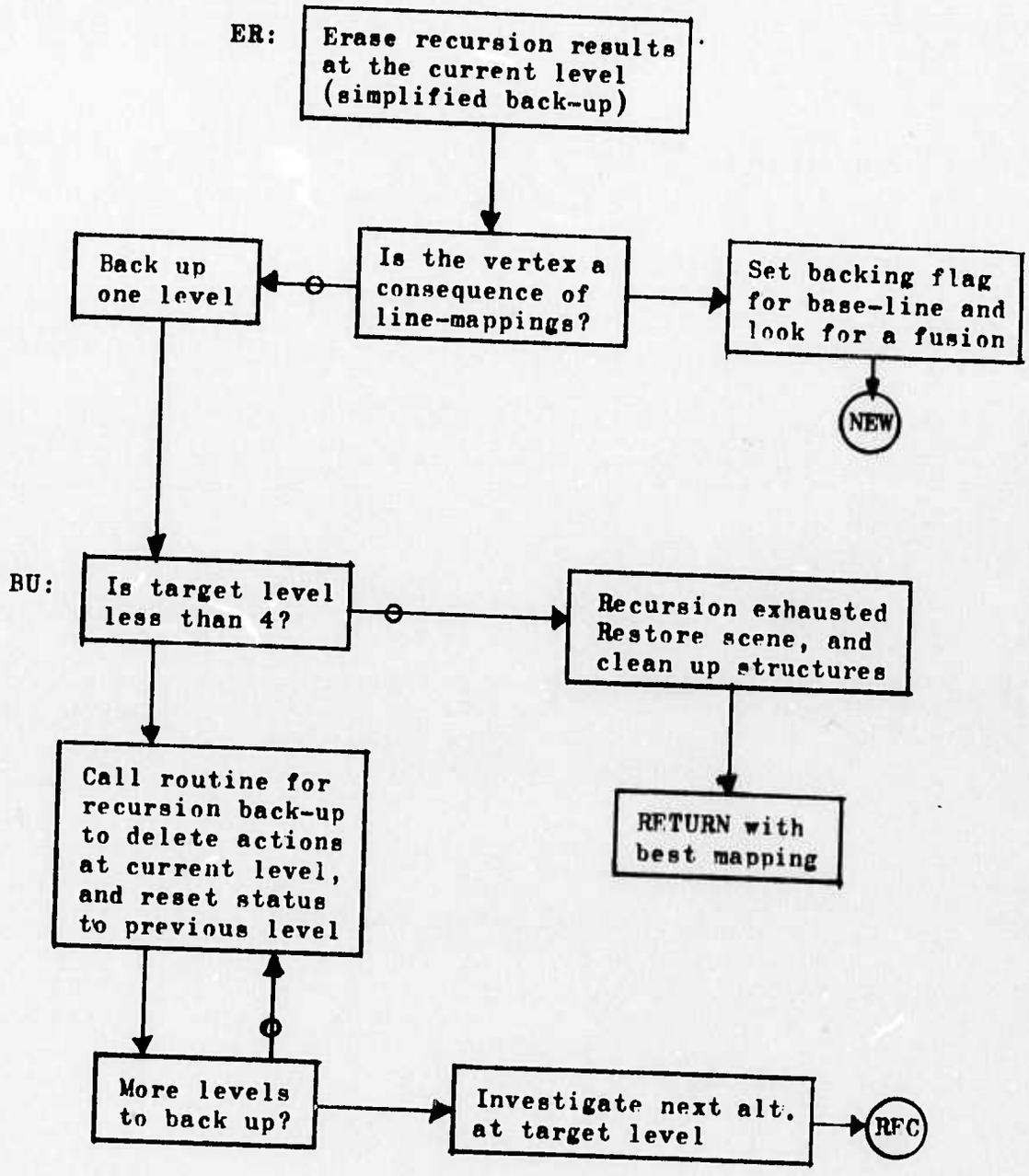


Figure 9.10

Erasure and back-up

and shows how the process ends, if a complete object hasn't been found before then.

The final diagram is in Figure 9.11, and it shows the deletion (back-up) of the actions at some recursive level, and of associated information. The back-up program has mechanisms taking care of collinearities, that is, of fusions and un-fusions of base-lines. If a consequence vertex is found, this routine backs up one more level, since it would be no use trying it again from another direction. This is the case also for a negative ray, for which an INCOV must have been attempted at some point in time.

It should be clear(er) now, how the recursive process makes use of prototype topology as well as line-feature information and equality classes (etc.) to provide guidance, in order to avoid much superfluous work, to direct back-up, and so forth.

The following section (a much shorter one) presents object completion, which could be a final process in this intermediate level system.

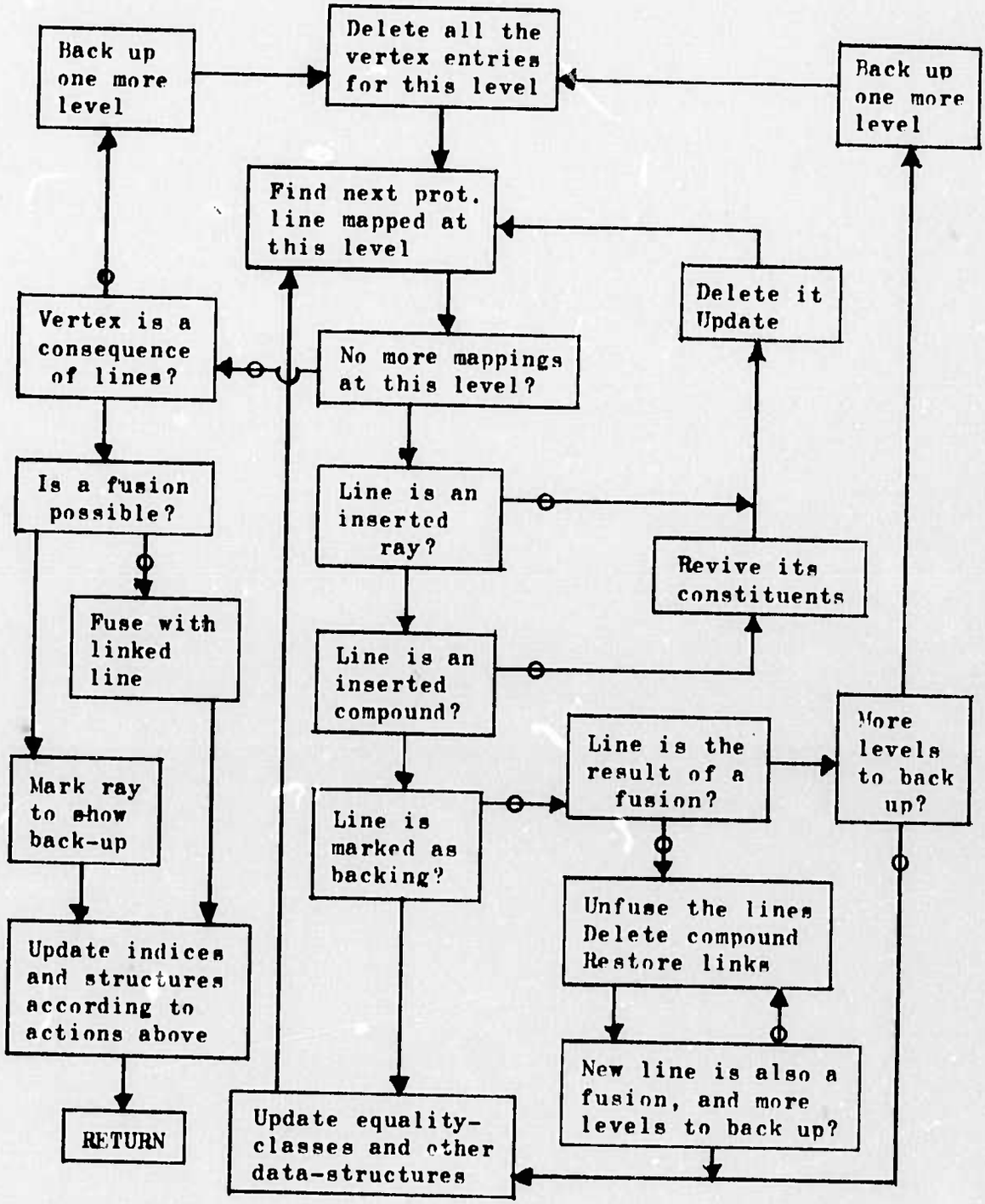


Figure 9.11

The main actions at back-up

10.0 OBJECT COMPLETION

THE BASIC IDEA:

The idea behind this phase is the following. Suppose the final scene (with all mapped objects removed) still contains some lines, and that some of the isolated objects are only partially mapped. It is logical in that situation to check and see whether some partial(s) might not be extended, or even completed, using those remaining scene-elements.

The concept is a very simple one, and so is the execution. We revive each partial (in order of decreasing complexity) and look for extensions and intersections of its rays (one-way mapped lines). Those are then tested in a process similar to the original mapping process.

The way this is done, practically, is simply as follows.
First we make sure the physical properties of the lines belonging to an object reflect the topology of that object. That is, the vertices are recomputed, and the line-coordinates are changed accordingly. This is done for all objects, whether completely or partially mapped.

Following this, the actual completion phase begins, and proceeds thus. The partial is brought out from the subconscious, and a new cross-reference and tentative-vertex evaluation is performed, this time with more liberal parameters, for instance allowing first intersections of pairs of unlinked lines regardless of distances. There is one important reservation here, namely that we do not allow extra lines to join the

fully mapped vertices of the object. We may safely allow relaxation of parameters at this point, for several reasons:

The partial will be put together as before, except possibly for new elements being linked to incompletely mapped parts of the object.
The scene is uncomplicated, having only comparatively few lines.
We are not dependent on the scene for a mapping key.
The mapping process ensures correct topology and feature consistency.

A new mapping is only attempted if there are changes in the connectivity of the object (due to the new cross-reference pass). Using a fully mapped line for the key, we call the mapping program, which returns with the best partial mapping, according to that new structure. This partial is at least as good as the original mapping, since the original will have been encountered during the matching. We compute the new or amended vertices, adjust the lines, and ship the object back into the subconscious.

This process continues, with the next partial, until either the scene or the subconscious is exhausted.

WHY NOT?

When this section was first written I had only done some preliminary work on implementing object completion. Having spent some more time thinking about these things, I decided that it might all be a bad idea. Let me explain...

First, experiments with many scenes have rarely produced cases where such a scheme would contribute to the performance of the system.

Secondly, it may well happen that spurious, irrelevant lines are absorbed into partial mappings, since linkages are less strictly required.

Thirdly, the elaborate heuristics for formation of tentative vertices, as well as the scheme for using partially similar features as keys, both contribute towards obviating the need for a specific object completion pass.

The last reason - a matter of policy - is that we do not strive to arrive at complete interpretations at any cost. If the scene is ambiguous or otherwise too difficult, we must rely on an extended scheme (such as proposed in Section 12) for further processing. Object completion belongs in that context, utilizing obstruction-, support-, and depth relationships. The present recording of the mappings (constituting scene interpretations) should prove well suited to the requirements of such extended schemes.

10.0

POST-PROCESSING

In order to show interpreted scenes more clearly, and to demonstrate the power of a knowledge-directed scheme, I could have added a hidden-line elimination phase. That process would be based on obstruction relationships, to be provided manually, lacking 3D knowledge and a support theory. Those concepts are no longer "intermediate-level", and one has to stop somewhere ...

The elimination of hidden lines or line-segments could be very straightforward. Basically, each line of an obstructed object would be intersected with the outlines of all obstructing ones, keeping the unobstructed segments.

I have resisted this temptation to produce good-looking final drawings, partly because I have had better things to do with my time, but mainly because such a program would not serve a useful purpose within the frame-work of the present system.

The only post-processing presently in this system is the completion of fully mapped parts of objects, according to the topologies of their respective matching prototypes.

By way of clarifying the concepts presented so far, and demonstrating the abilities (and weaknesses) of this intermediate-level vision system, we now give some typical examples of scenes and their analysis.

11.0 EXAMPLES - RESULTS - DISCUSSIONS

11.1 COMMENTS

First some general comments. All examples of system performance in this report represent scenes of uniformly coloured (whitish) objects, which are not unrealistically ideal, in asmuch as they are fairly beat-up, having been manhandled (and kicked around?) by many people since they were made [Falk 1970].
The scene background is always a black cloth covering the table-top. (Feel free to regard this as cheating!)

In most of the examples normal office-type lighting (over-head, diffuse) was used, otherwise the auxiliary (diffused) light sources surrounding the "Hand-Eye Table". Needless to say none of the examples have been in the least edited, nor are they a non-typical, selective sample of scenes that work especially well. They also all use standard parameter settings. Finally, some scenes were created by people other than myself.

The pattern of presentation of any given example is the logical succession starting with the TV-image, going through pre-processing, looping through object mapping and isolation (showing isolated object and amended scene each time), and finally presenting the interpreted scene as a conglomerate of partially or fully mapped objects.

The examples are commented as needed, to focus attention on points of particular interest or typicality. The section will end with a discussion of results and system performance, treating fortes as well as shortcomings.

11.2 EXAMPLES OF SYSTEM PERFORMANCE

The example that has been presented in parallel with this unfolding story provides instances of shadows, broken lines, occluded vertices, missing lines, double lines, and a split object. The initial line-drawing and the final scene interpretation are reproduced here for convenience (Figure 11.1).

There were good keys into all of the objects, and the matching program was able to find complete mappings in all cases. We note how essential the line-fusion heuristic was here, in establishing the lower vertices of the large body, which had keys only to the top part. The same heuristics were instrumental in finding the long horizontal object, and the object in front.

The wedge presented no great problems, only extrapolated vertices. Note also how double lines are removed with an object when they may be assumed to be caused by it (judging by closeness and parallelity to object lines).

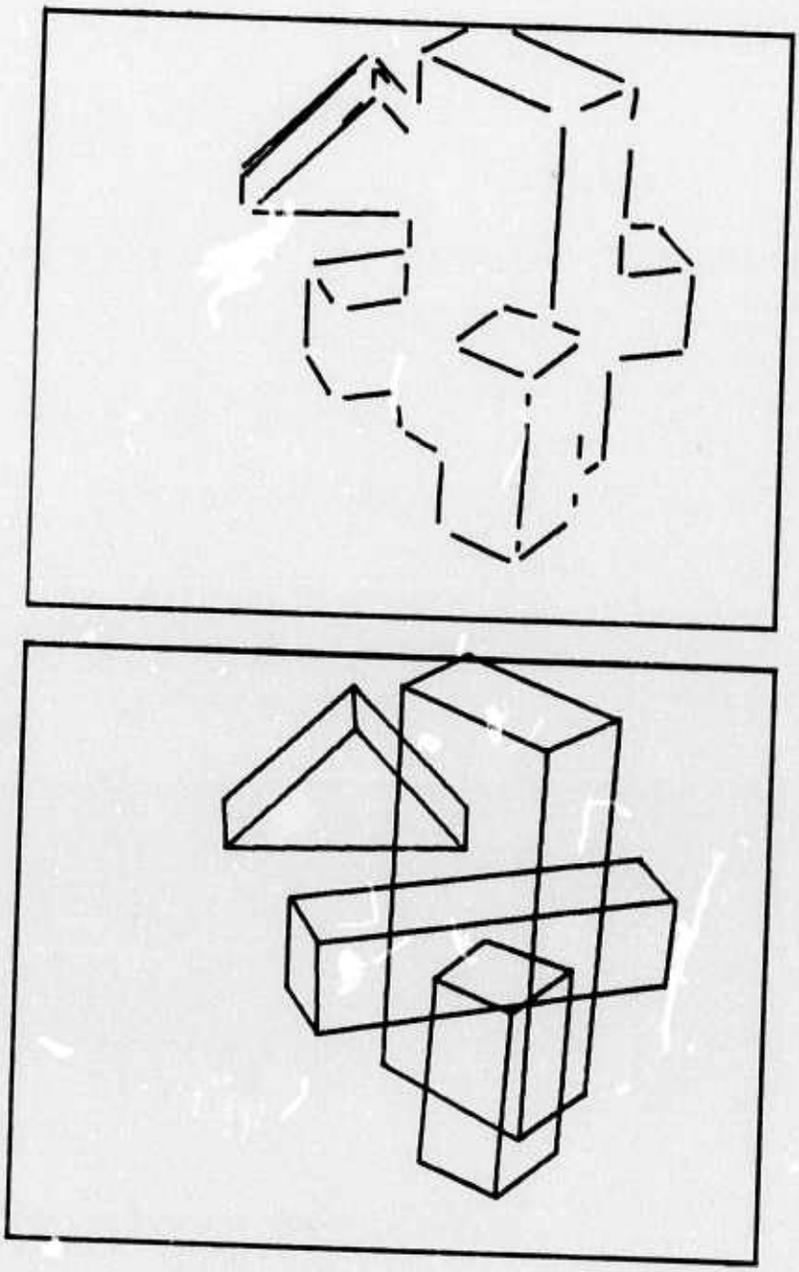


Figure 11.1

SC10: Initial lines - Final interpretation

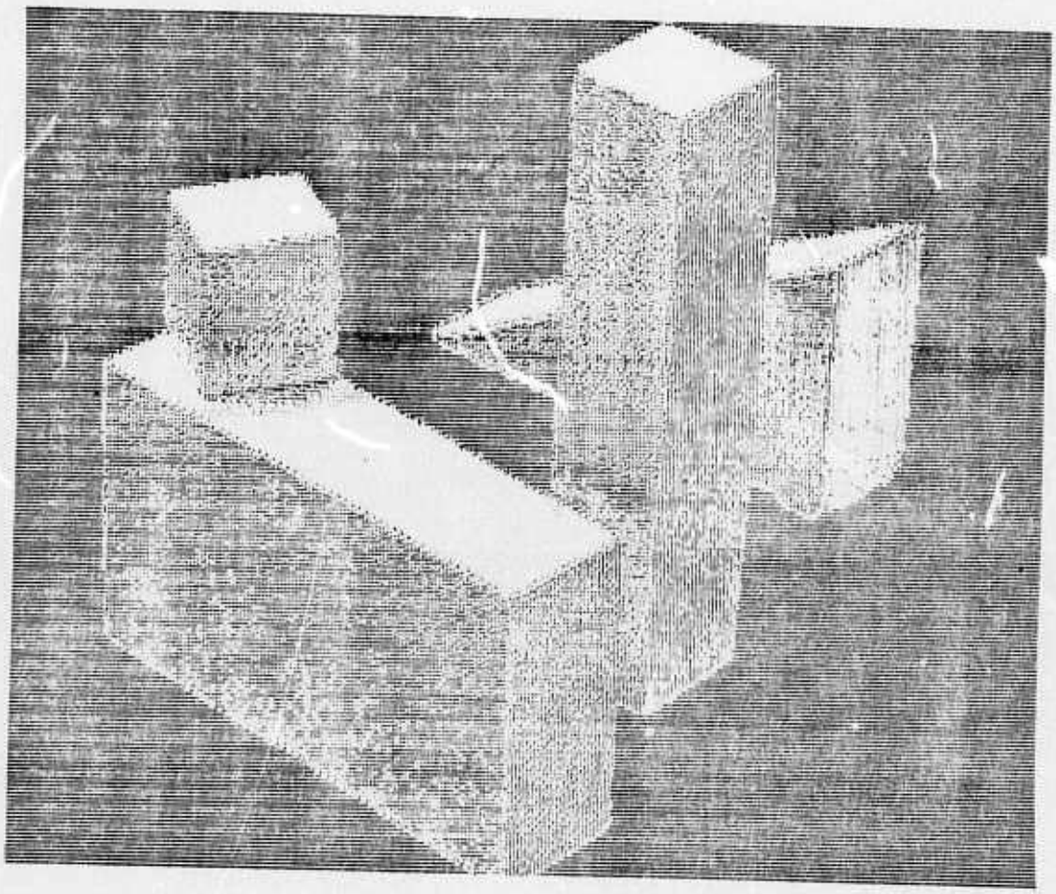


Figure 11.2

SC11: TV-image

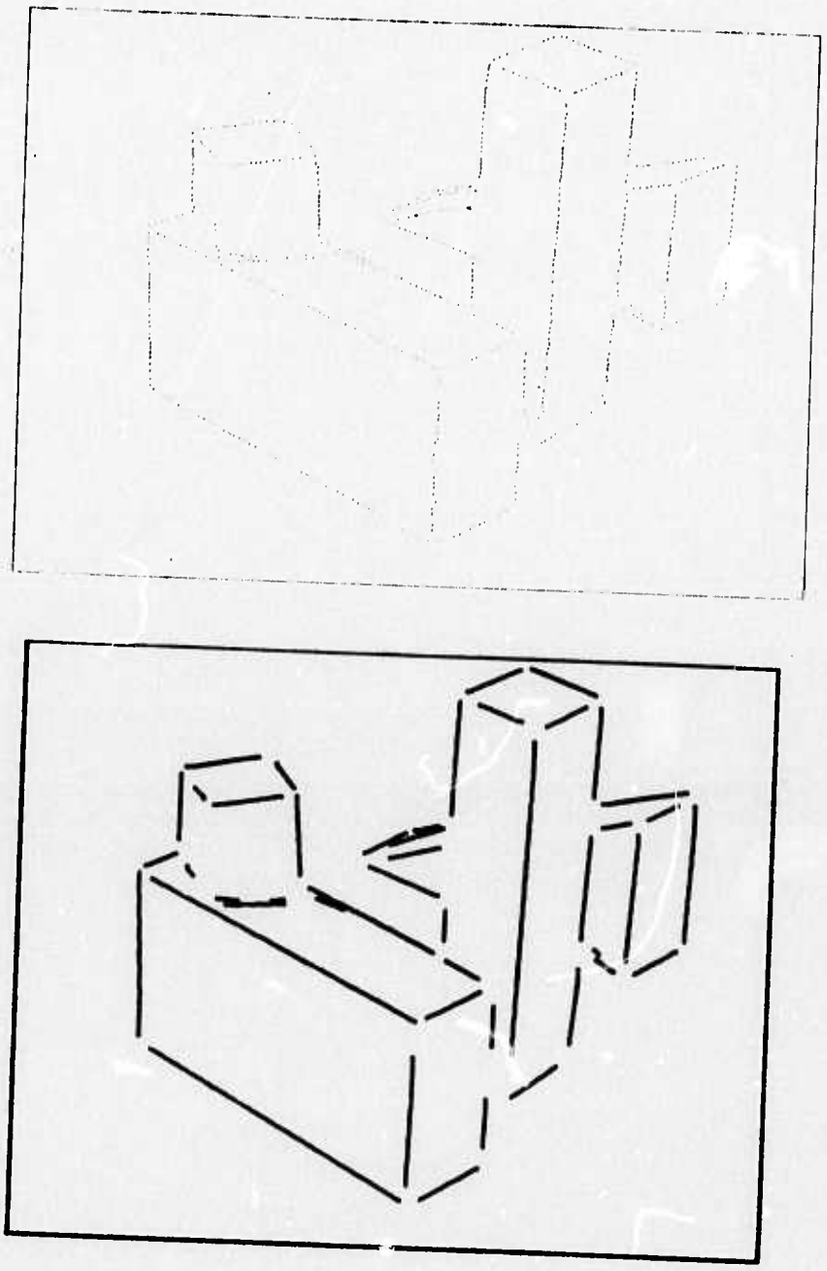


Figure 11.3

SC11: Edge-data - Initial lines

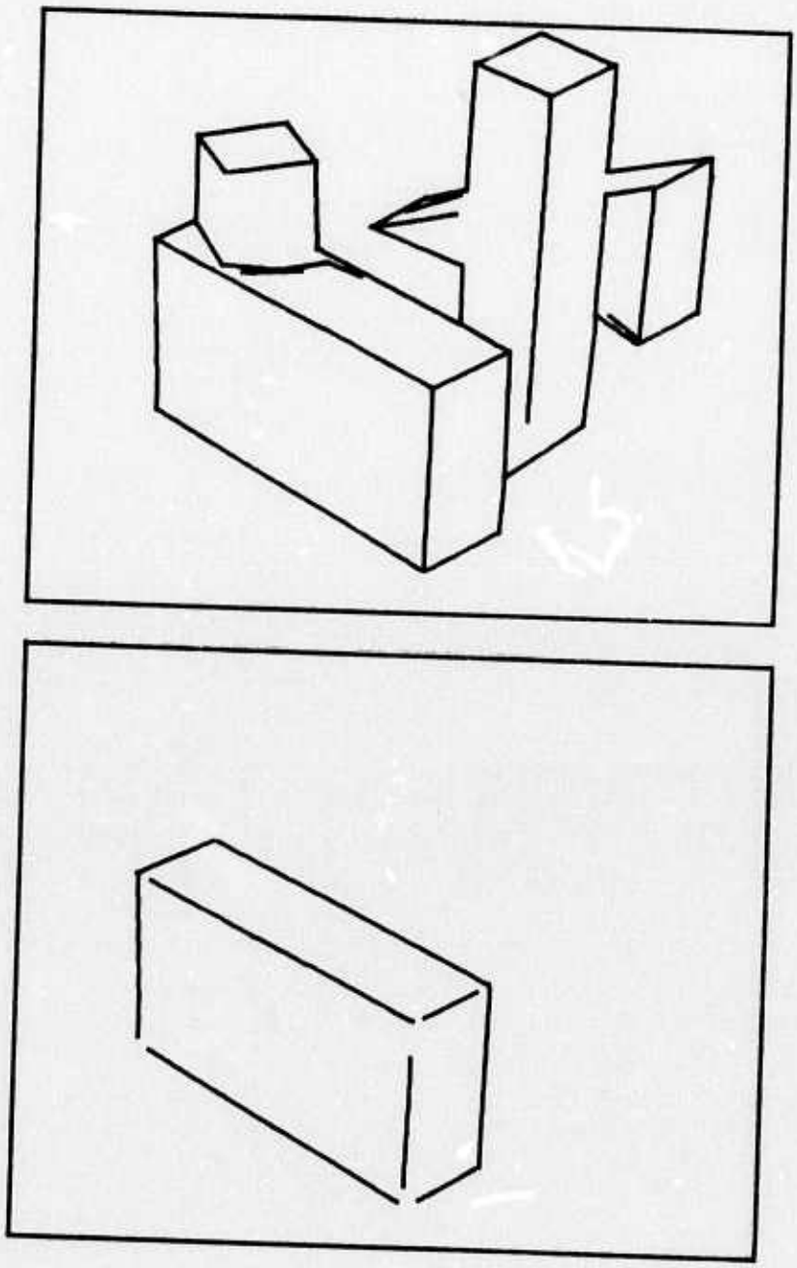


Figure 11.4

SC11: Tentative vertices - First isolation

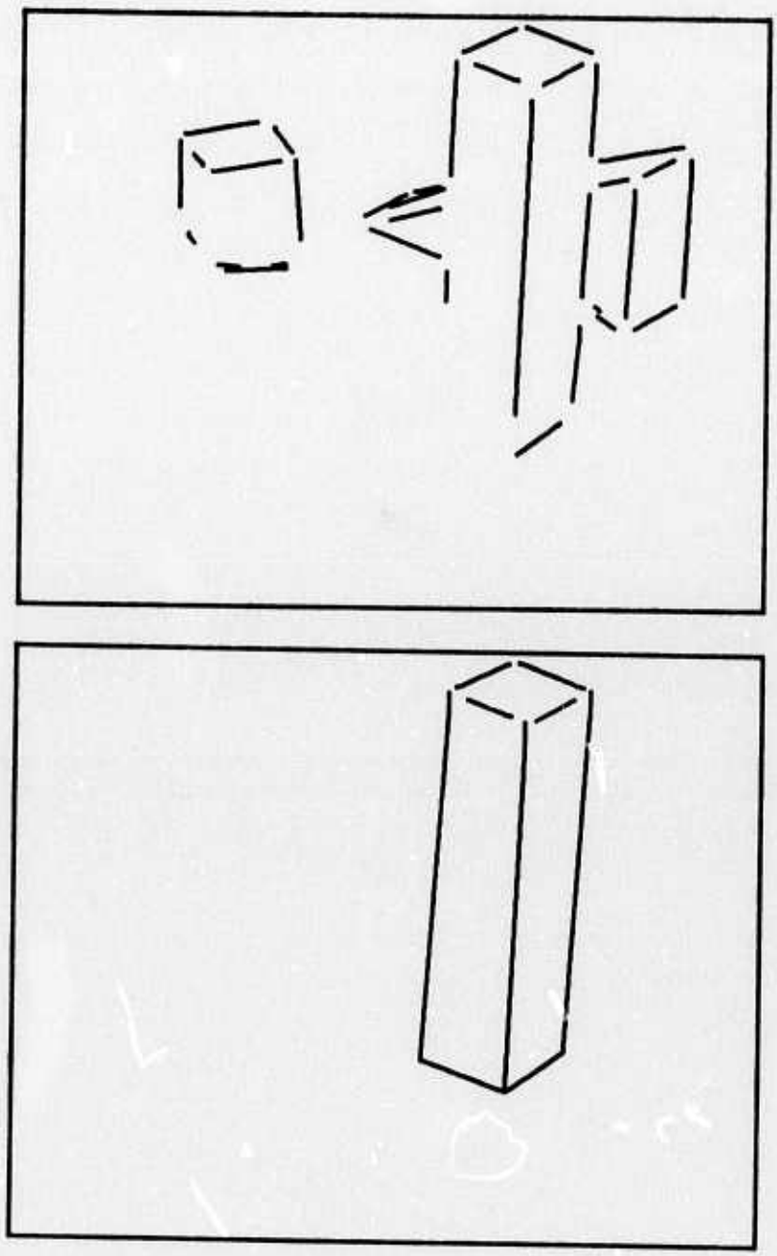


Figure 11.5

SC11: Amended scene - Second object

11.2

The next scene (Figure 11.2, Figure 11.3, Figure 11.4, Figure 11.5, Figure 11.6, Figure 11.7, and Figure 11.8) has a shadow in front of the small cube, which makes that dimension hard to determine, and the final object is slightly distorted here. The glare-line at the left end of the wedge presented no problem - the feature-guided fusion mechanism continued past it in search of the right kind of vertex. The line could not be assumed to be a part of the final object, and it was left as garbage.

The scene is finally parsed correctly, with some distortion.

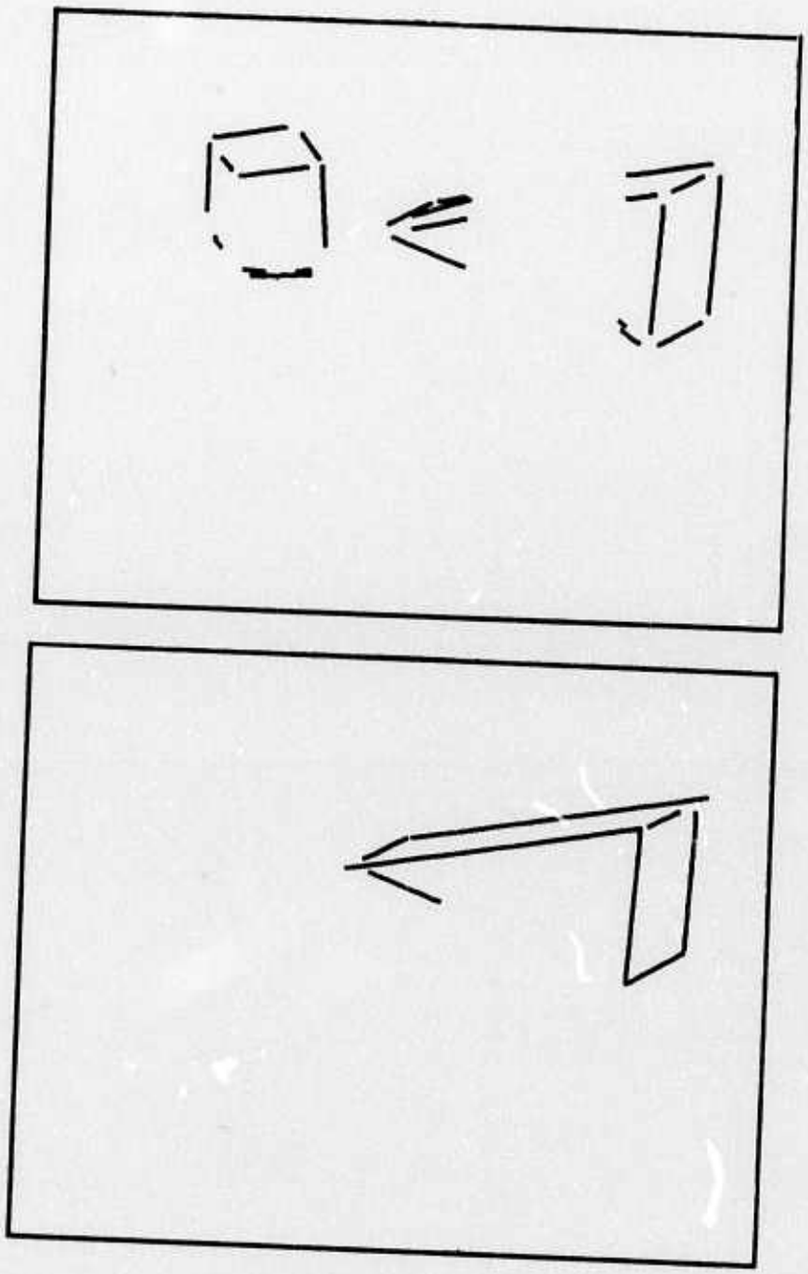


Figure 11.6

SC11: Amended scene - Third object

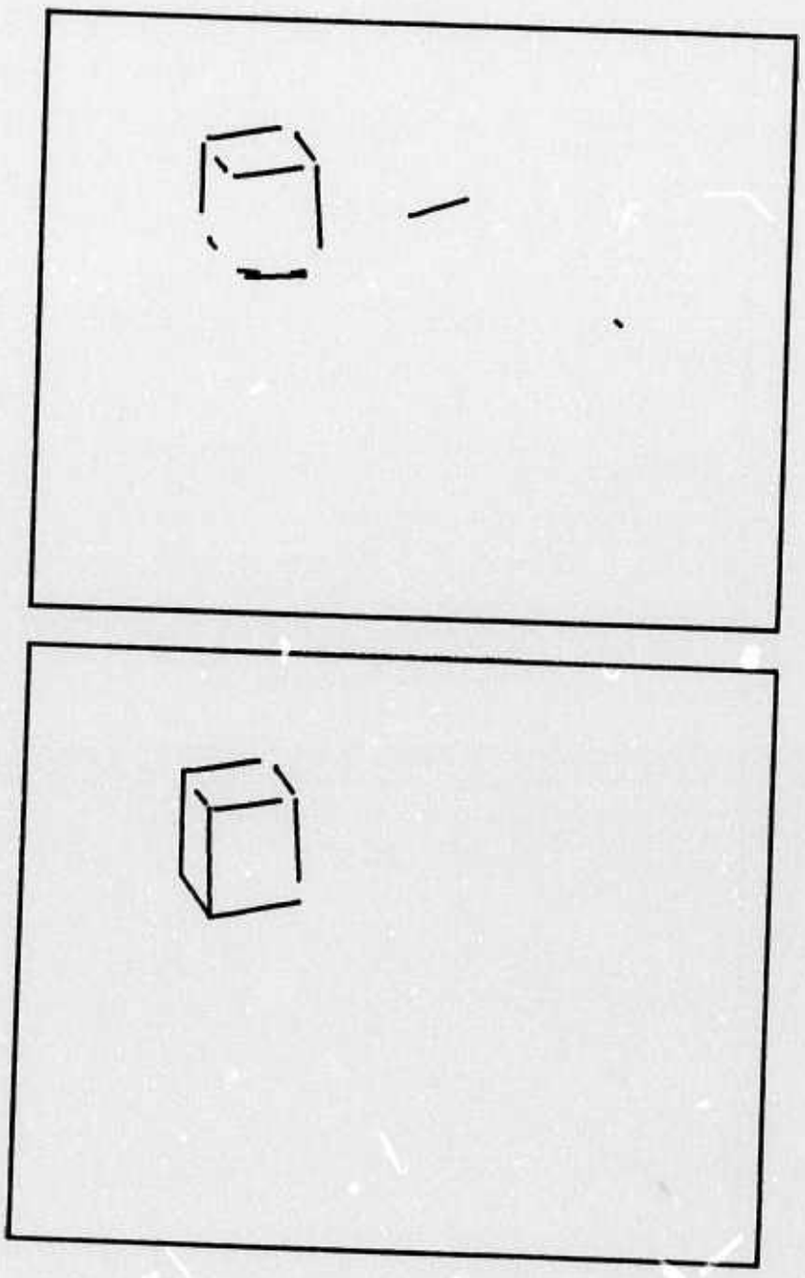


Figure 11.7

SC11: Amended scene - Fourth object

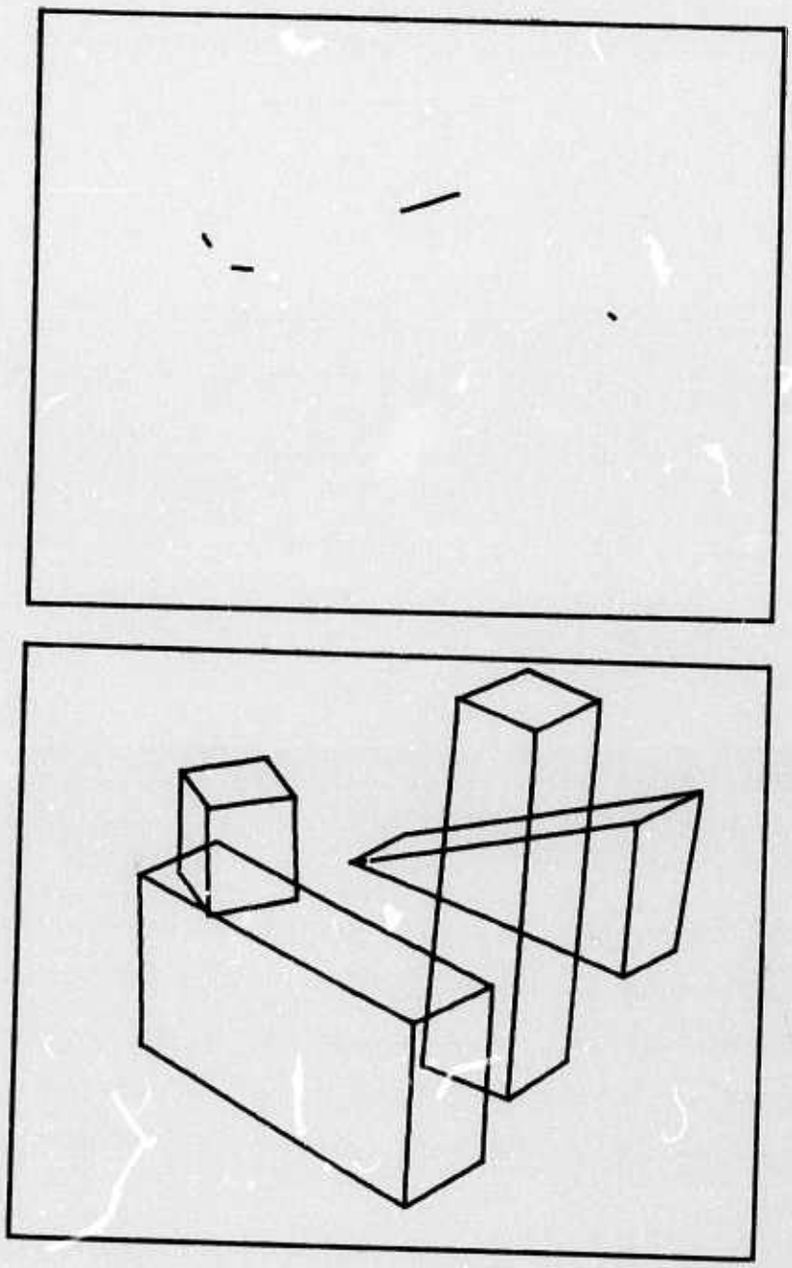


Figure 11.8

SC11: Amended scene - Final interpretation

The third scene presented here has one area completely messed up by both shadows and glare, namely the left side of the right-hand parallelepiped. Both the TV-picture (Figure 11.9) and the edge- and line-drawings (Figure 11.10) as well as the tentative vertex connectivity (Figure 11.11) show the effects thereof.

The shadow on top of that object, caused by the reclining beam, gives rise to a very specific problem in the extraction of the latter object (Figure 11.12). The lower right-hand vertex of that beam does not get connected properly, only through a short segment in-between. That vertex, then, is found by the matching heuristics as an intersection consequence vertex, defined by an inserted ray (from the top) and an existing ray (from the right). At that point, the feature template demands an inserted ray, pointing to the left. That ray is found to be collinear (and is therfore linked) with the bottom line of the beam, and that object may finally be completed.

After easily extracting the wedge (Figure 11.13), the remaining object is the shadowed parallelepiped. The center vertex is established by two existing rays, but the three vertices on the left outline of the object are hypothesized on the basis of the intersection-consequence heuristic, using inserted rays when necessary. This is shown in Figure 11.14.

Hence this scene is finally interpreted correctly (Figure 11.15).

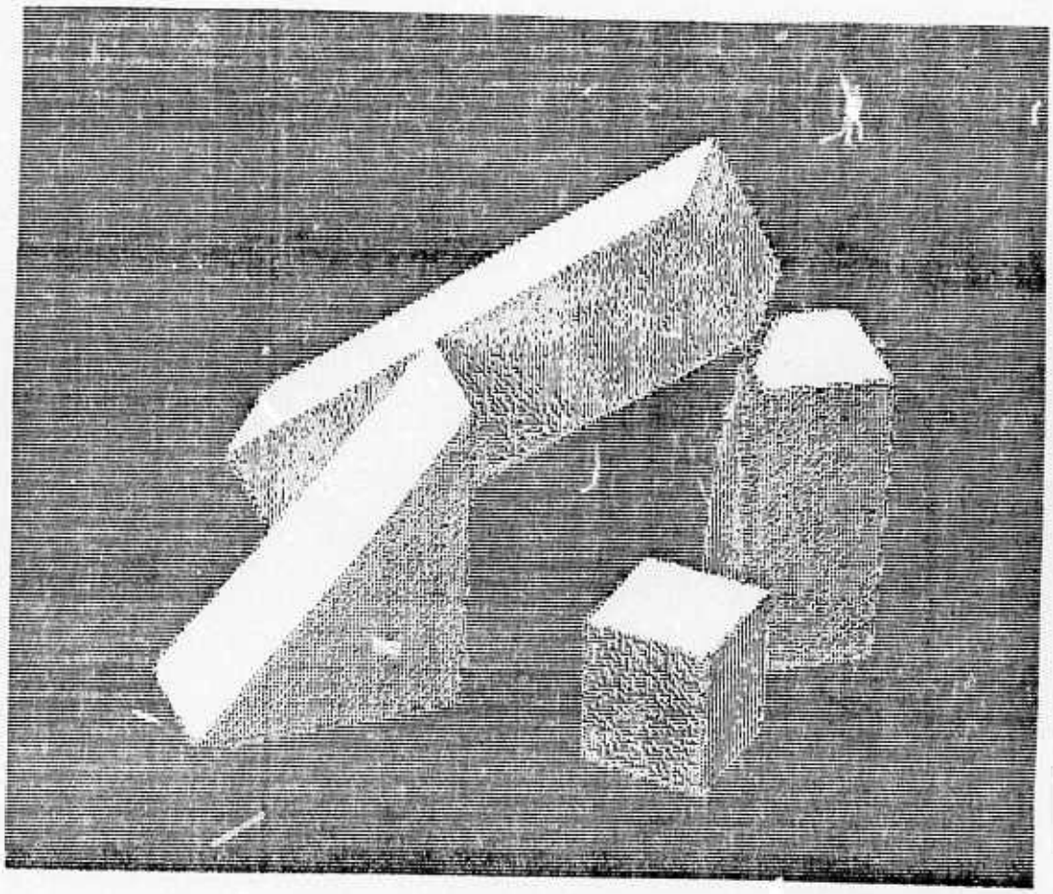
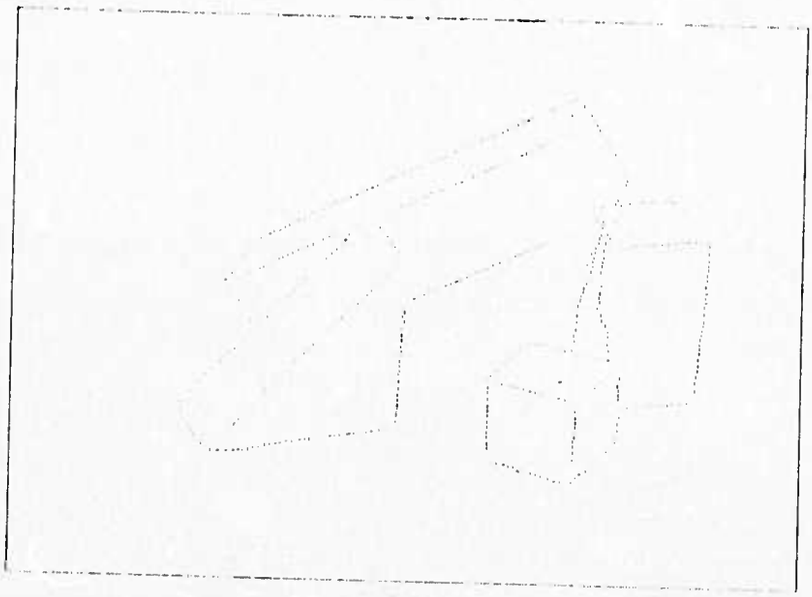


Figure 11.9

SC12: TV-image



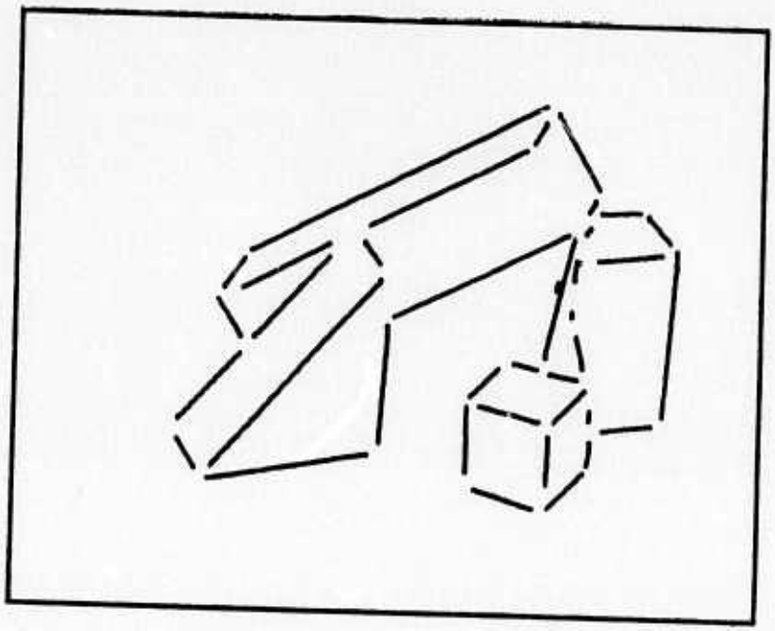


Figure 11.10

SC12: Edge-data - Initial lines

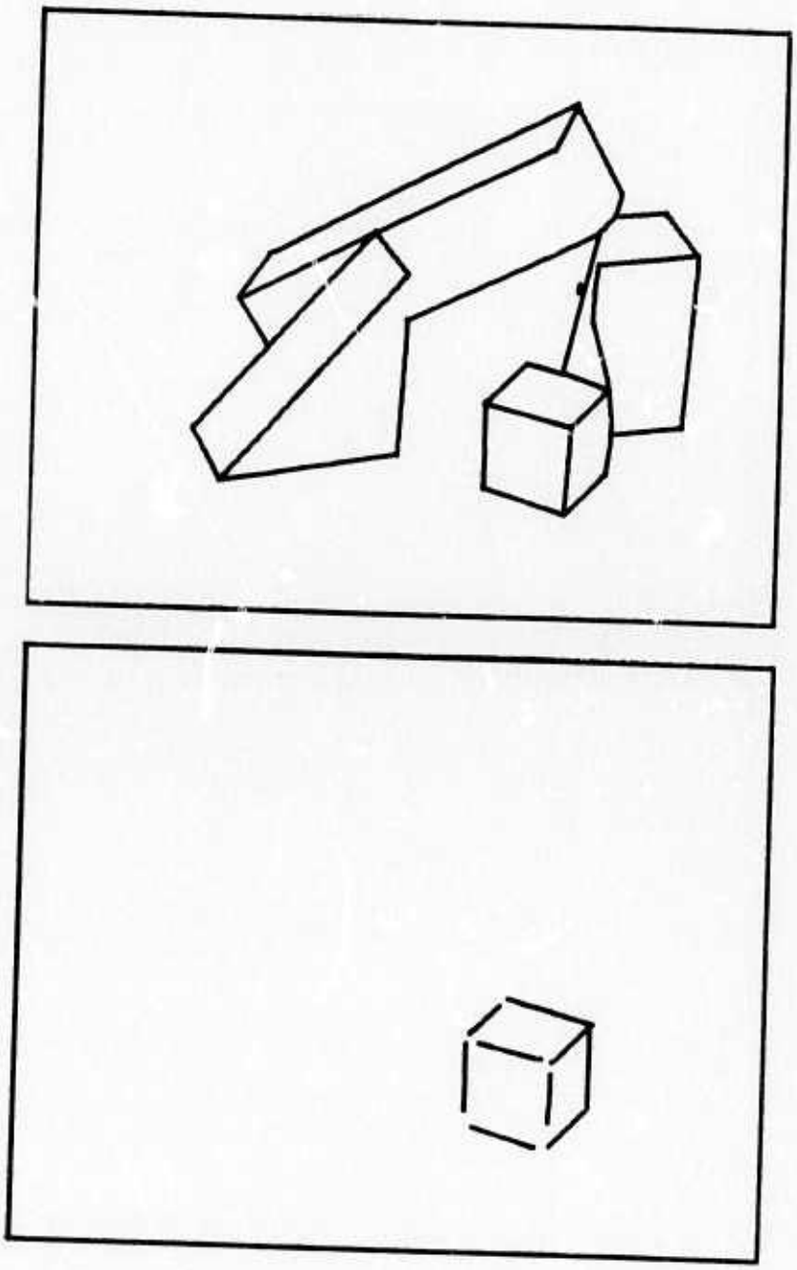


Figure 11.11

SC12: Tentative vertices - First object

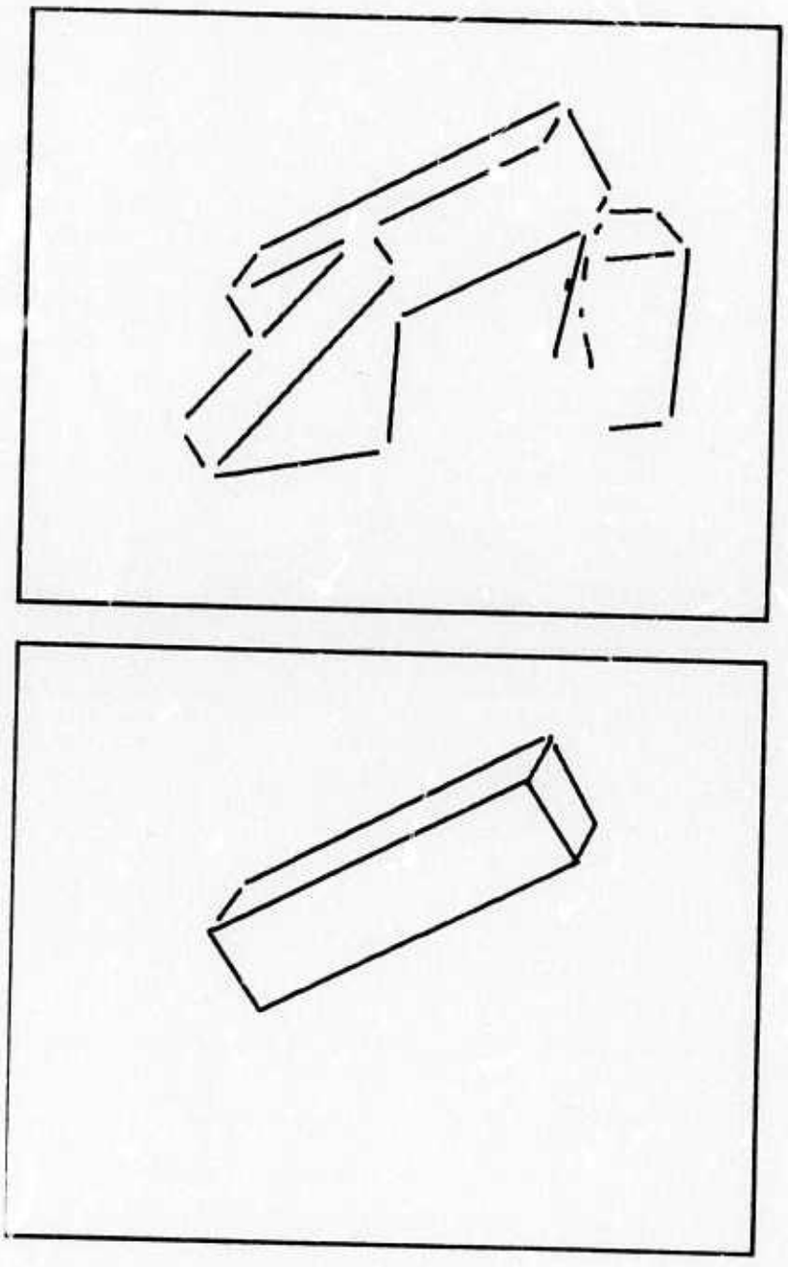


Figure 11.12

SC12: Amended scene - Second object

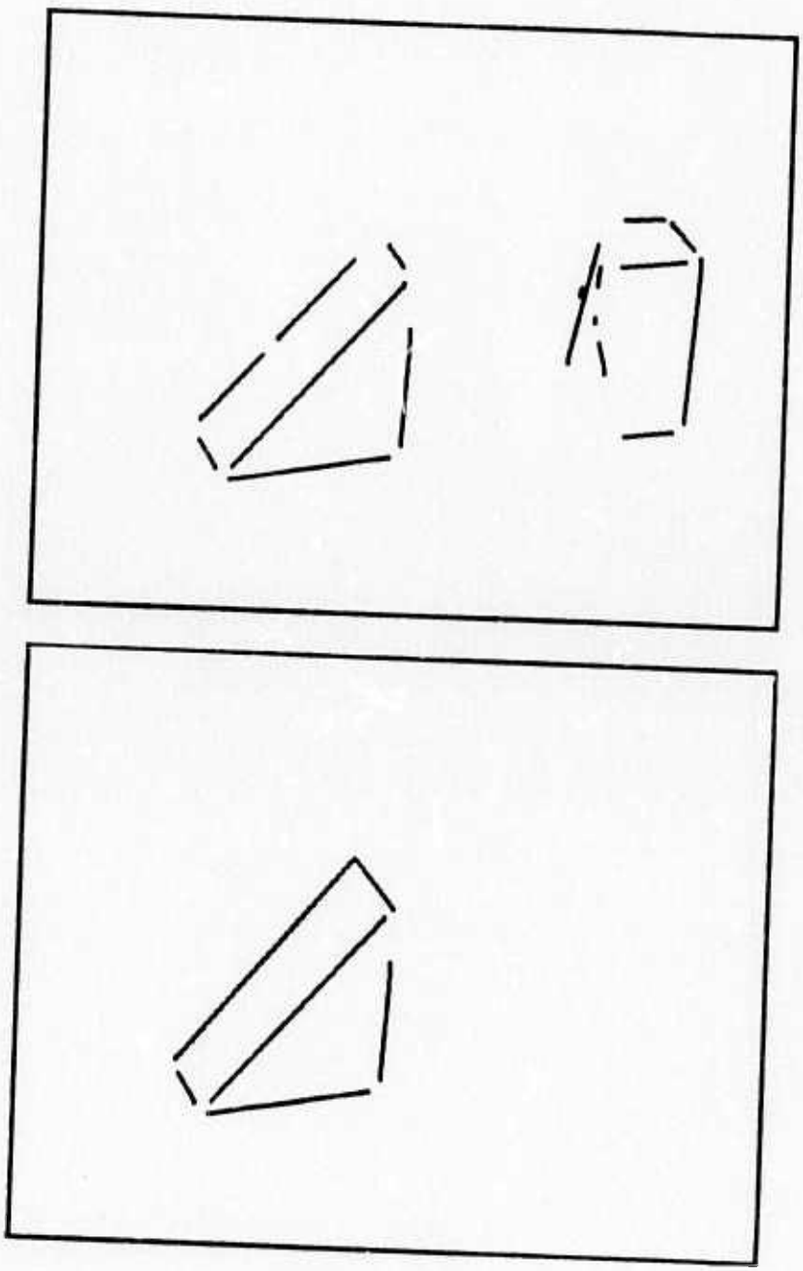


Figure 11.13

SC12: Amended scene - Third object

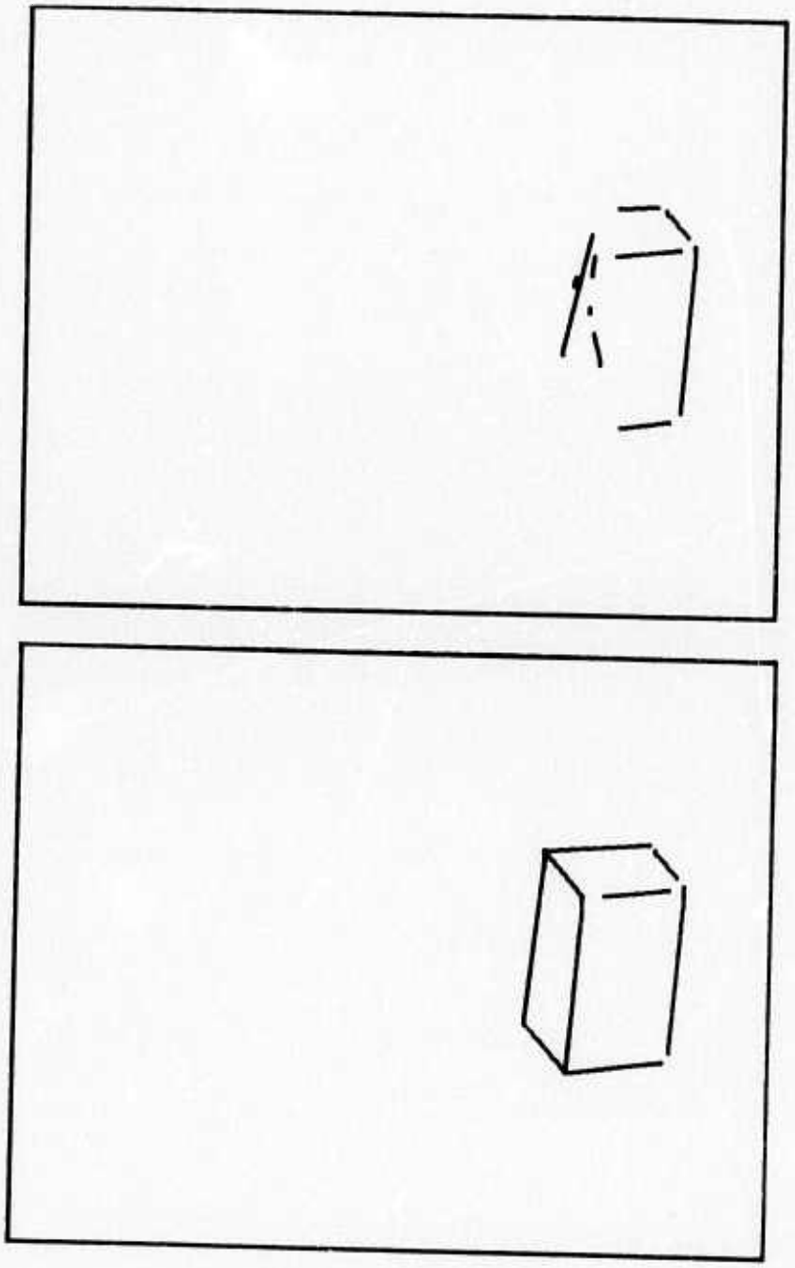


Figure 11.14

SC12: Amended scene - Fourth object

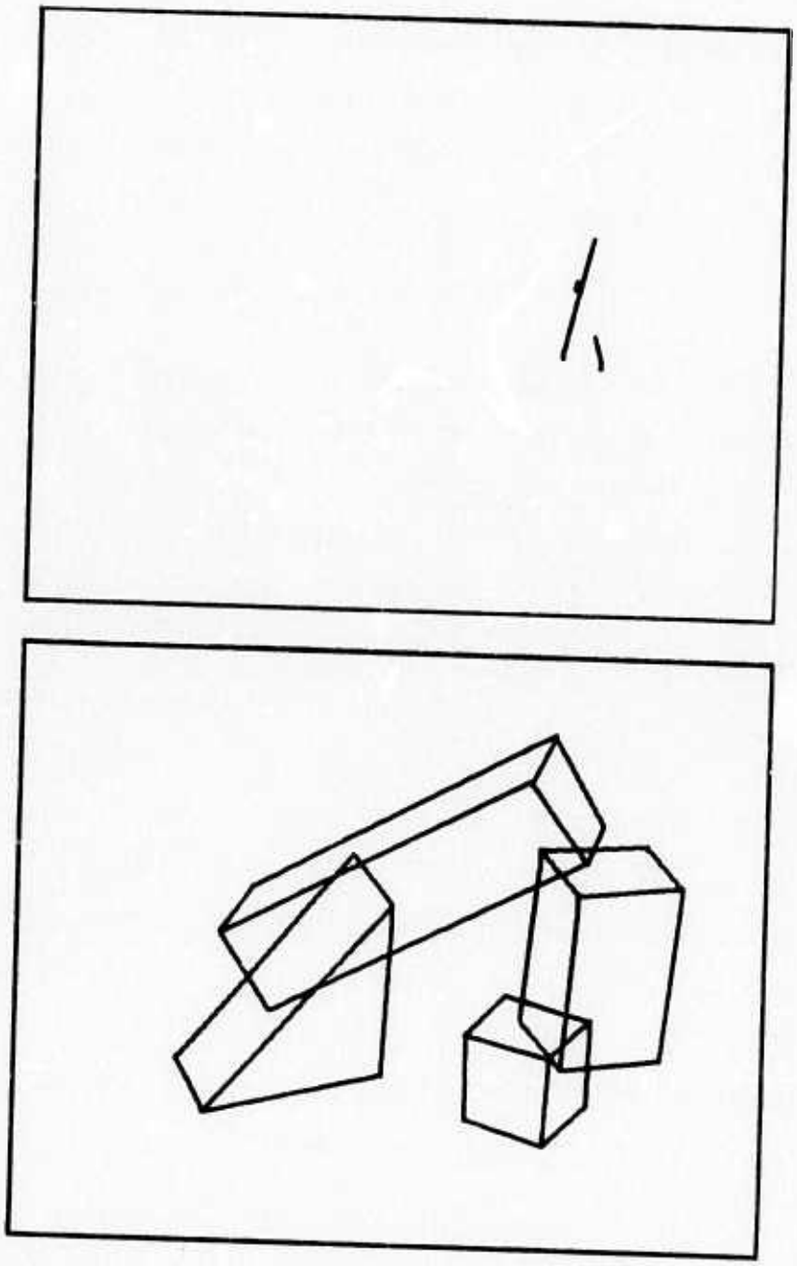


Figure 11.15

SC12: Amended scene - Final interpretation

The fourth scene is slightly more complex, in that it contains five objects (TV-image in Figure 11.16, edges and initial lines in Figure 11.17), but it presents no difficulties we haven't encountered in previous examples, including coincidental line-vertex alignments (Figure 11.18), which are a source of sadistic delight to the template-driven matcher.

The small cube is first to go, an easy match. There are also very good keys into the large wedge, which follows next (Figure 11.19). The top parallelepiped is then severed from the cube and the small wedge, completed and isolated, taking the double line with it (Figure 11.20).

The wedge is the next object to be extracted (Figure 11.21), with ample usage of fusion-, insertion-, and intersection-consequence heuristics.

The amended scene (Figure 11.22, top) shows a cube with two false vertices (top-left), which are however easily discarded by the feature-template, parallelity-class, and length-class heuristics, so that the cube may be extracted in perfect shape.

The shadow-lines are left as garbage, as shown in Figure 11.23, which also presents all of the objects superimposed.

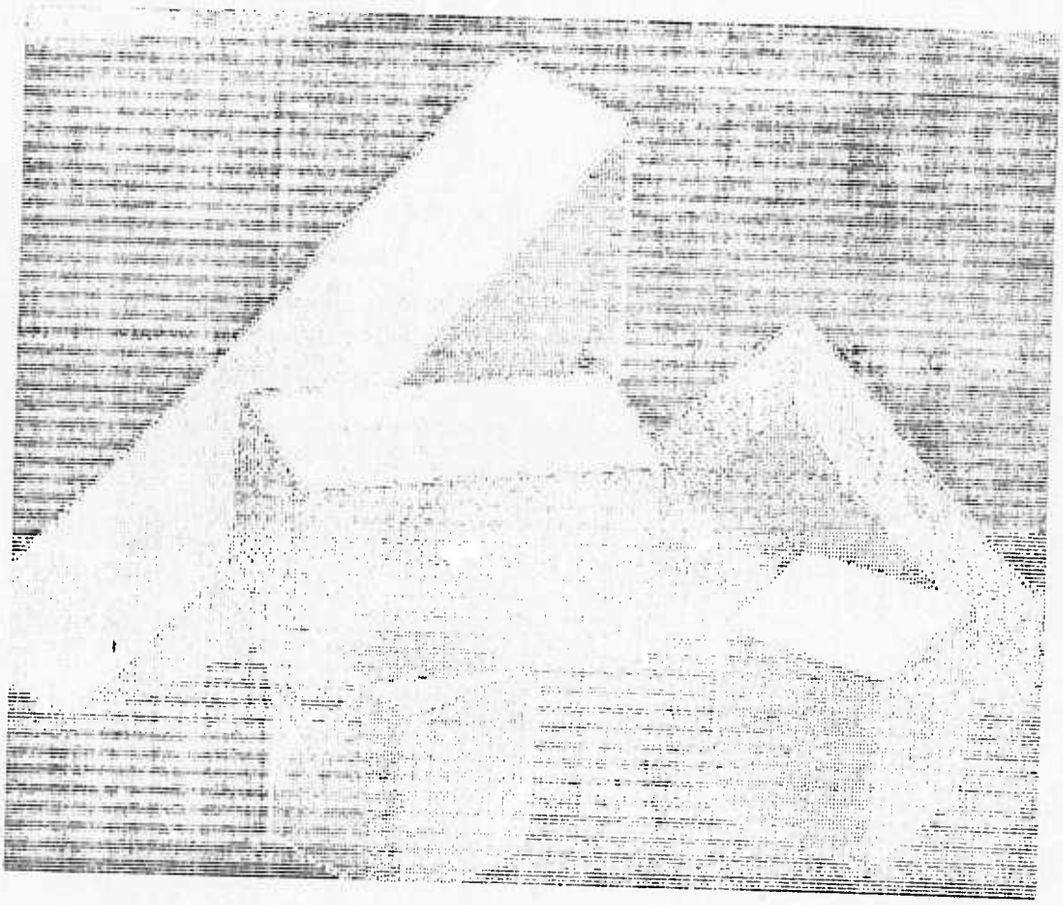


Figure 11.16

SC3: TV-image

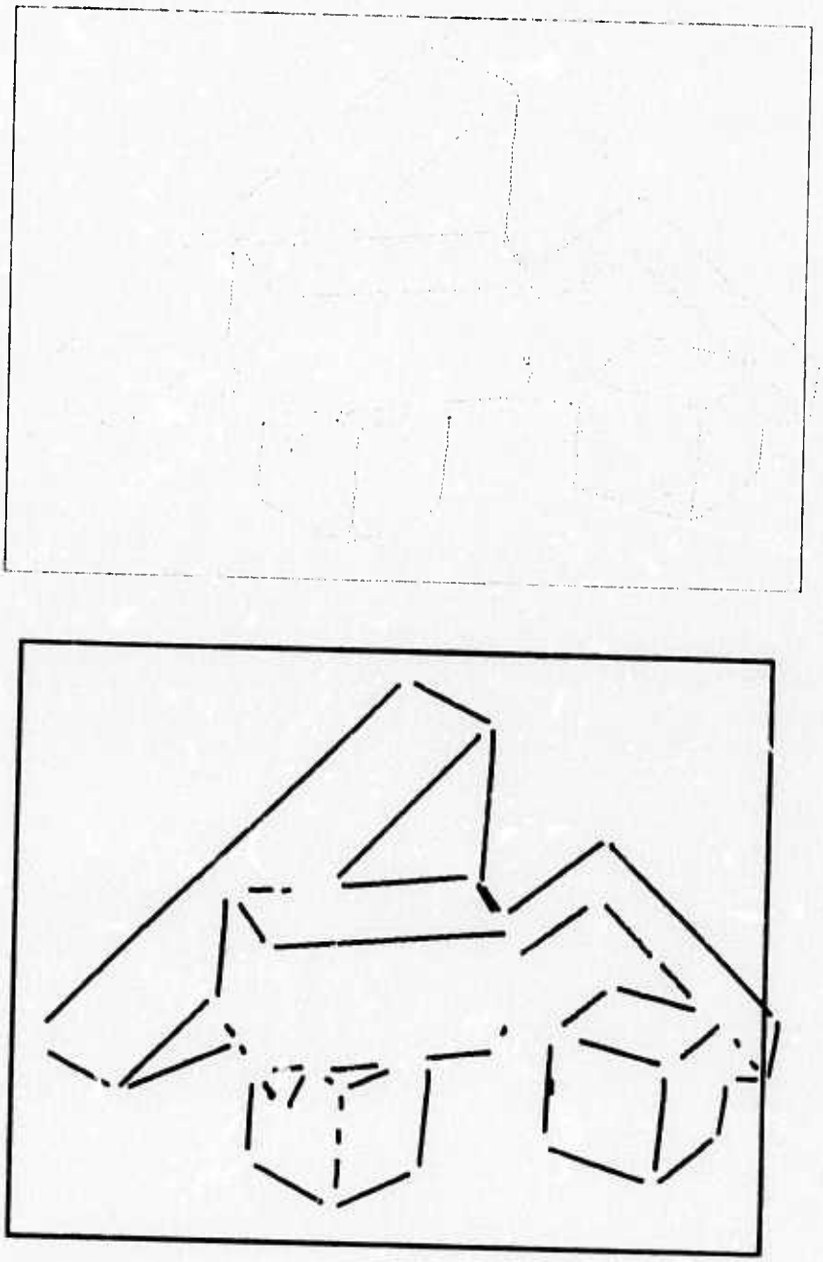


Figure 11.17

SC3: Edge-data - Initial lines

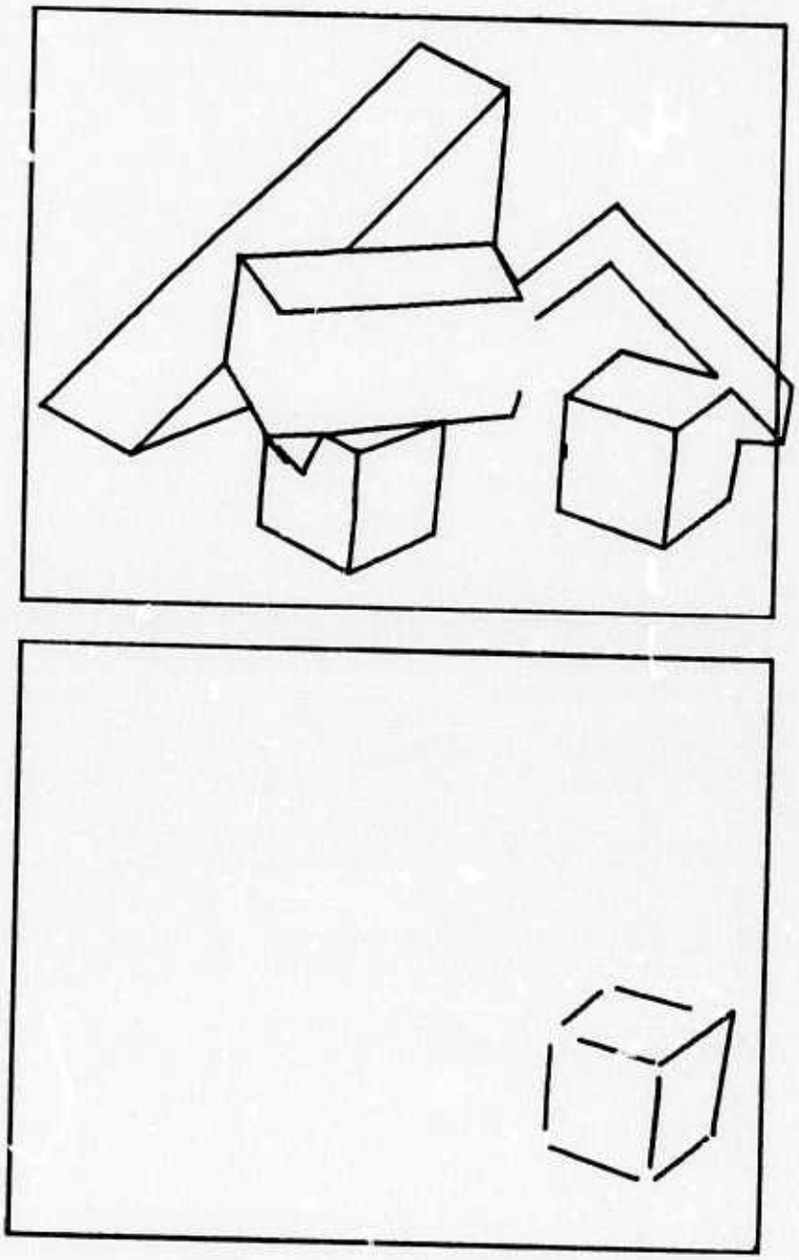


Figure 11.18

SC3: Tentative vertices - First object

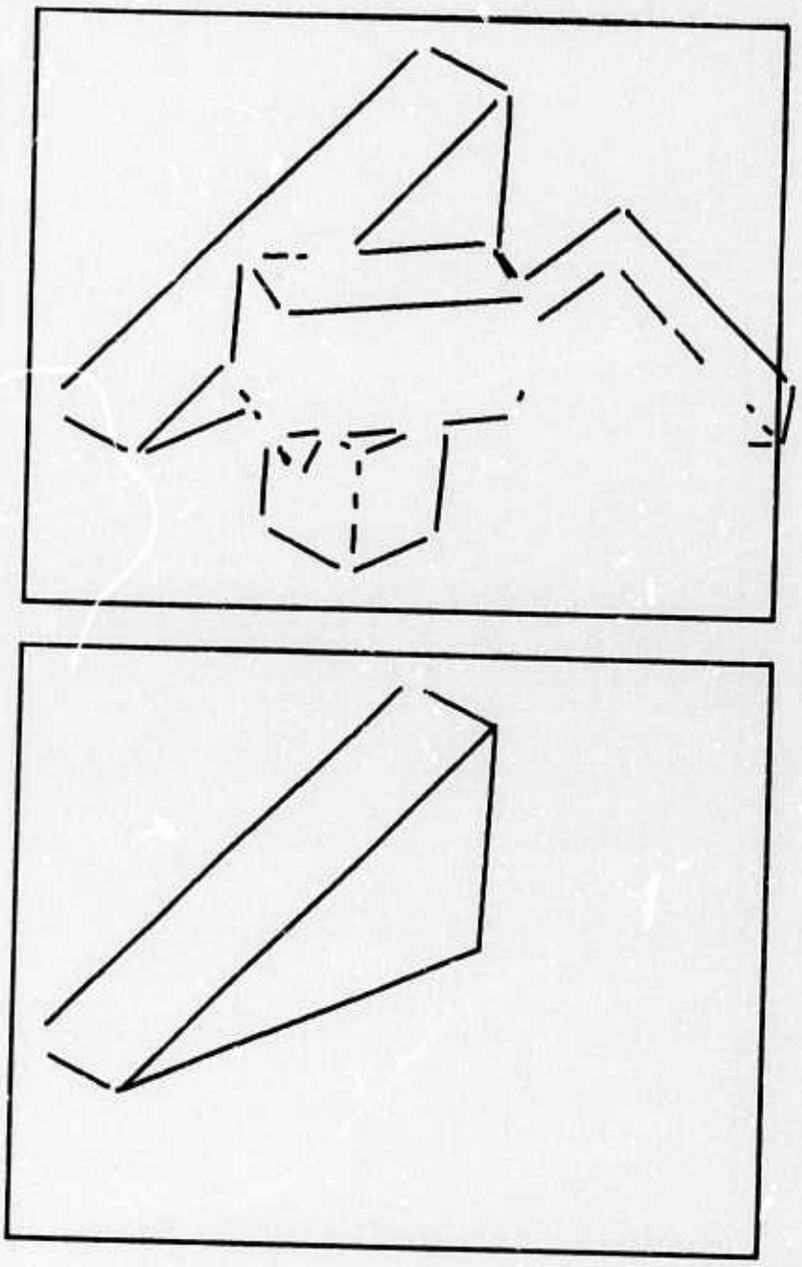


Figure 11.19

SC3: Amended scene - Second object

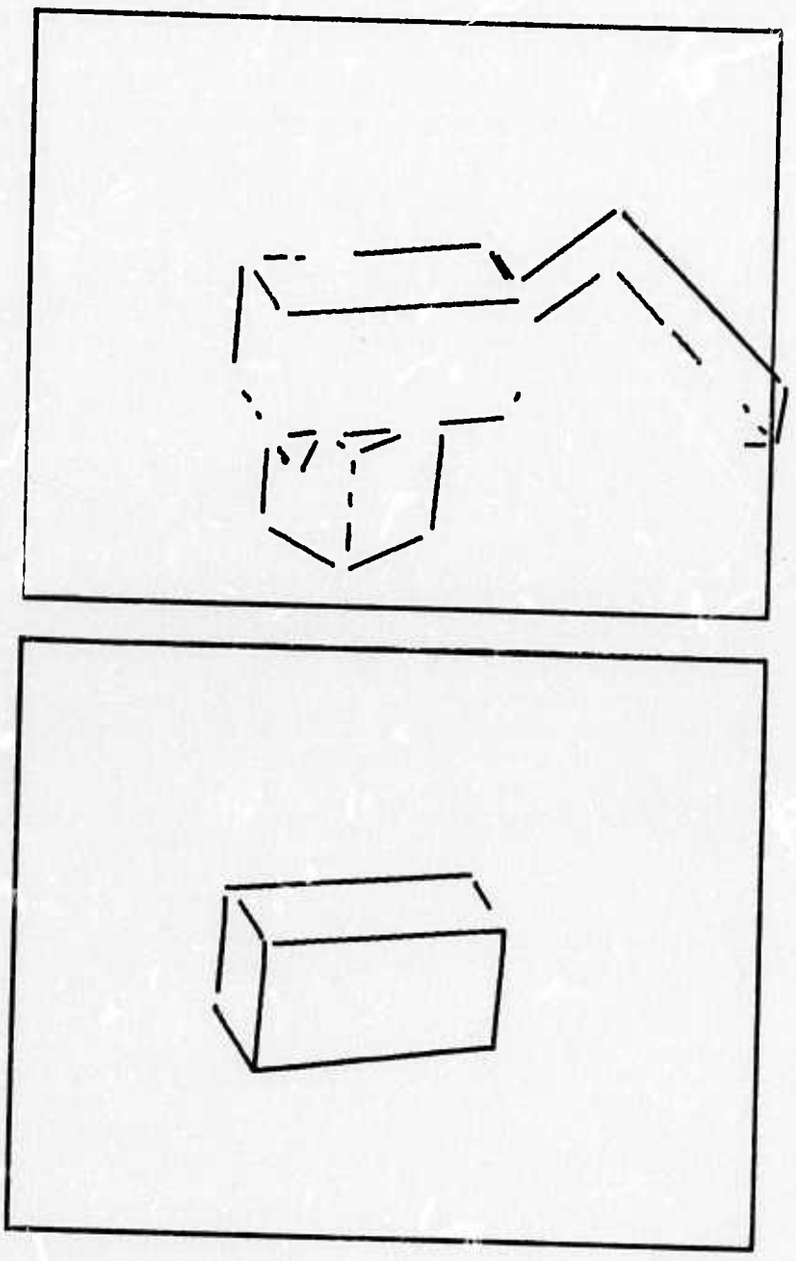


Figure 11.20

SC3: Amended scene - Third object

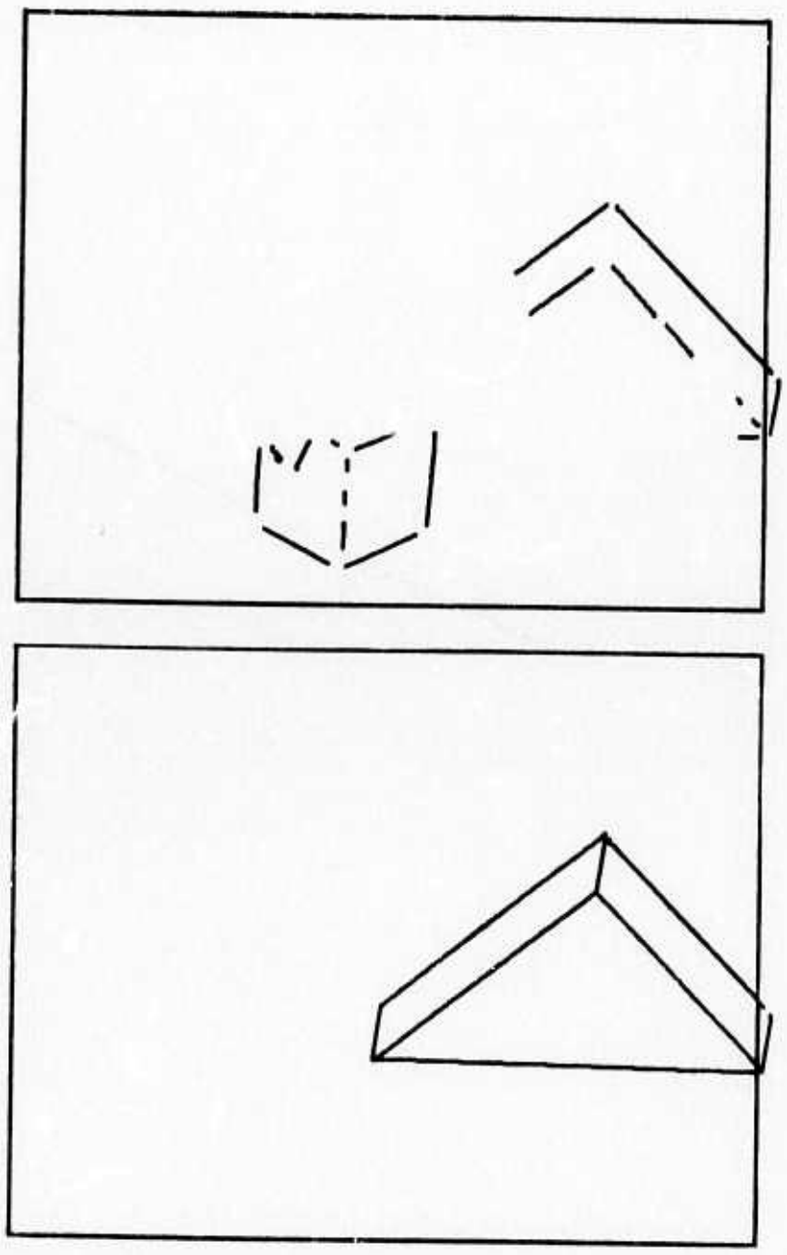


Figure 11.21

SC3: Amended scene - Fourth object

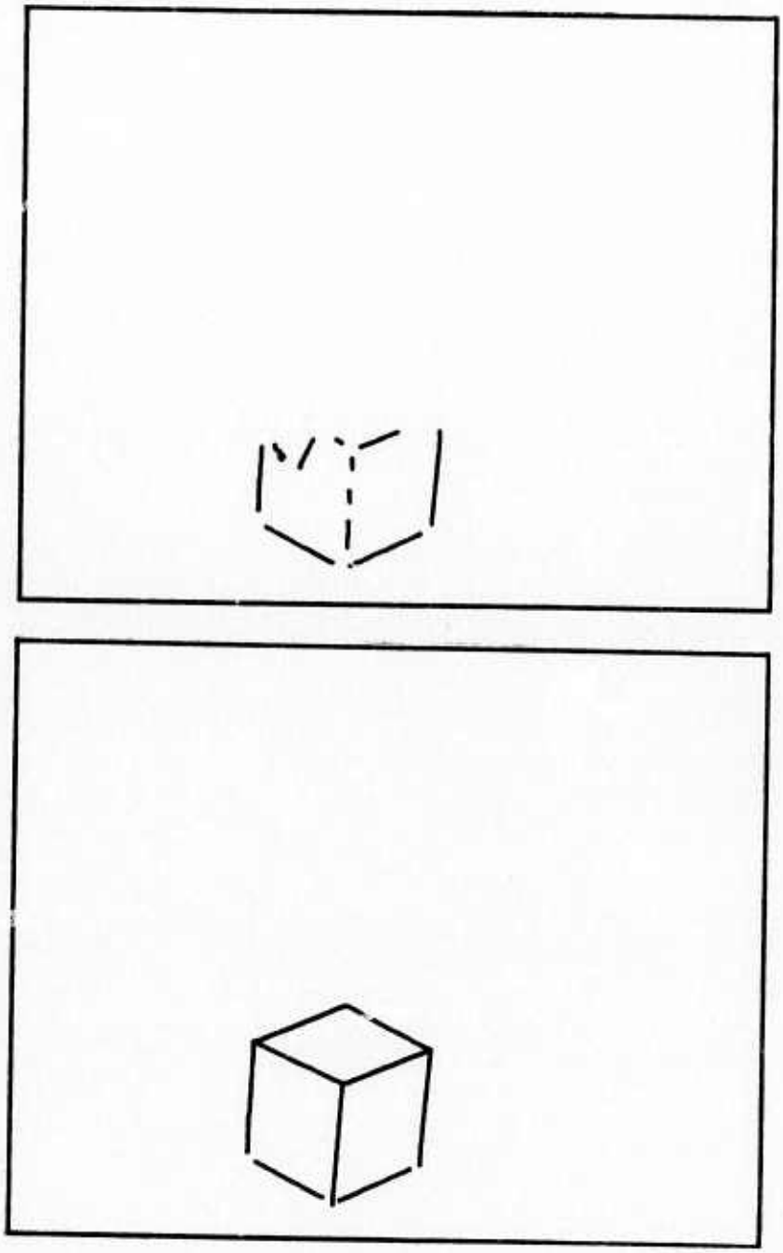


Figure 11.22

SC3: Amended scene - Fifth object

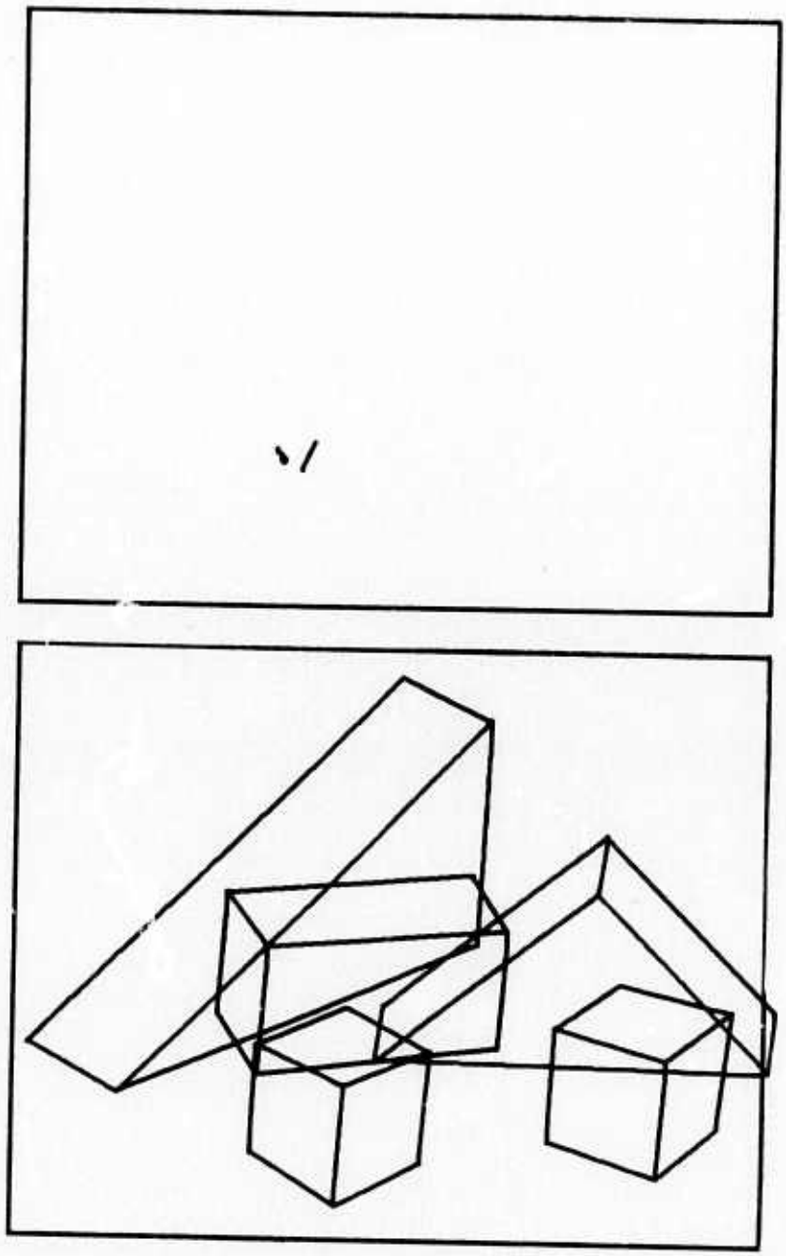


Figure 11.23

SC3: Amended scene - Final interpretation

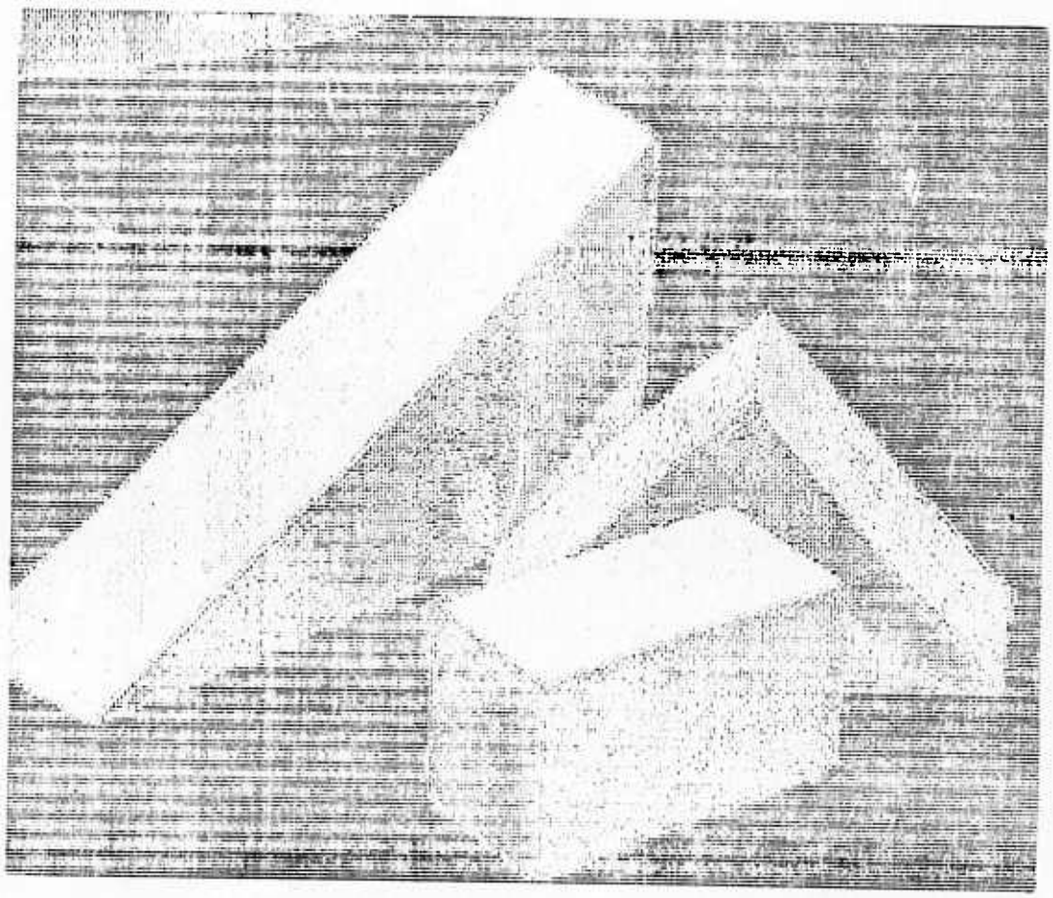


Figure 11.24

SC2: TV-image

The fifth example (TV-image in Figure 11.24) looks misleadingly simple, at least compared with our previous examples. However, there are a couple of subtle problems involved in parsing this scene. Looking at the edge-drawing and the initial line-drawing in Figure 11.25, we note the presence of a short line-segment at the lowest vertex of the large wedge, and that the long bottom line of that body is not connected at its right end. This gives rise to a tentative, somewhat narrowed wedge, which is however discarded in favour of the correct one.

Figure 11.26 shows the first match, a parallelepiped.

The correct match for the second object (the wedge mentioned above), shown in Figure 11.27, uses the long bottom line rather than the short segment. The connected drawing (top of Figure 11.26) indicates why those two alternatives are investigated (connectivity of left lower vertex).

It may be interesting to see some of the contenders for this second object, and I have included two figures containing alternative (but not as good) matches, namely Figure 11.28 and Figure 11.29. The first of those contains the narrow wedge I just mentioned, the second (top) a partial wedge with the triangular face on the right.

Now there is only one thing left in the scene, a wedge. The edge-drawing (Figure 11.25) clearly shows that one short interior line-segment is misdirected. This state of affairs gives rise to an alternative, shorter wedge, which is eventually discarded for the better match shown in Figure 11.30.

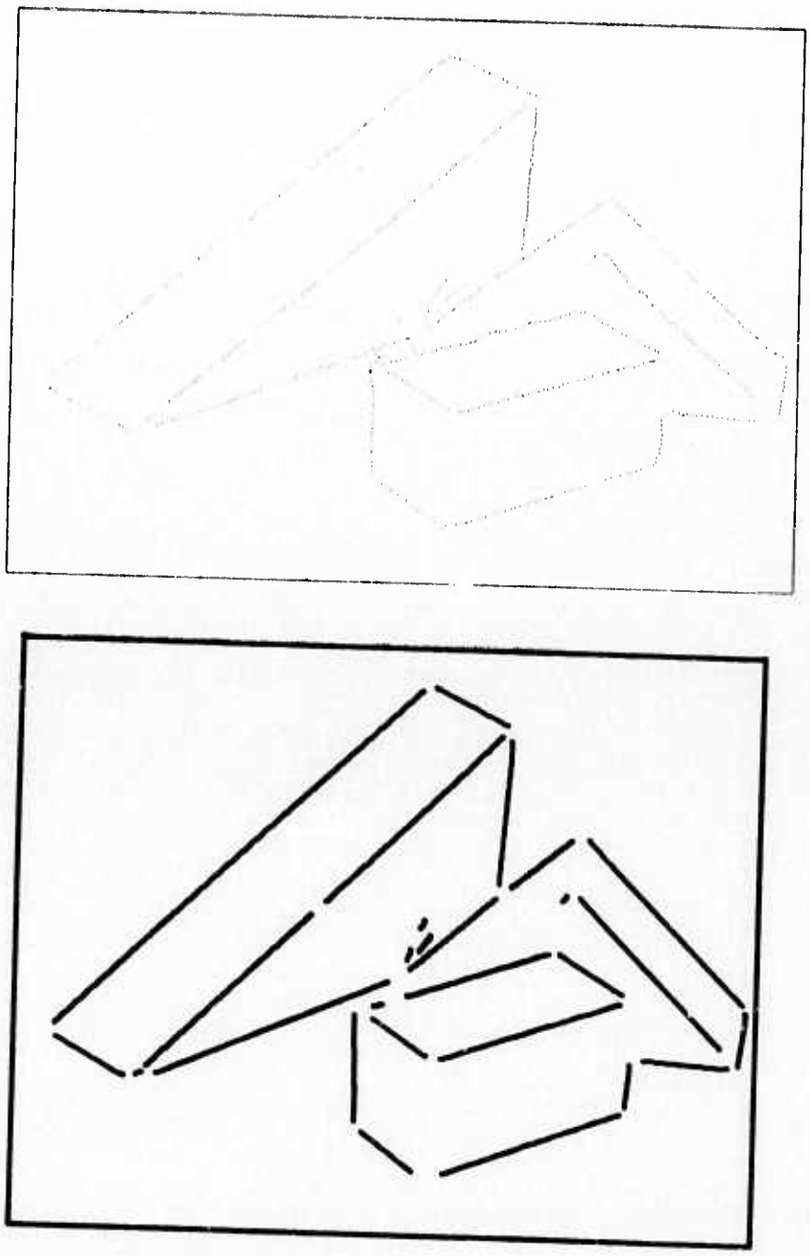


Figure 11.25

SC2: Edge-drawing - Initial lines

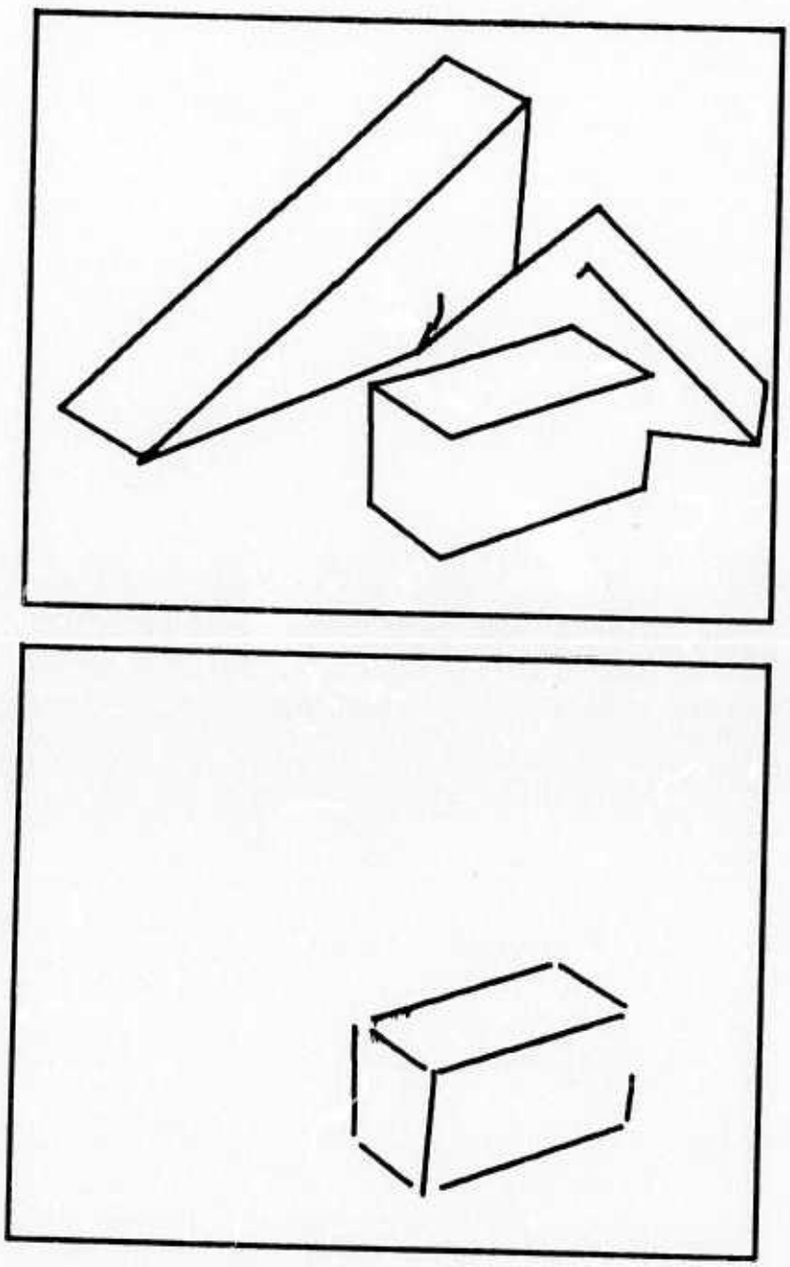


Figure 11.26

SC2: Tentative vertices - First object

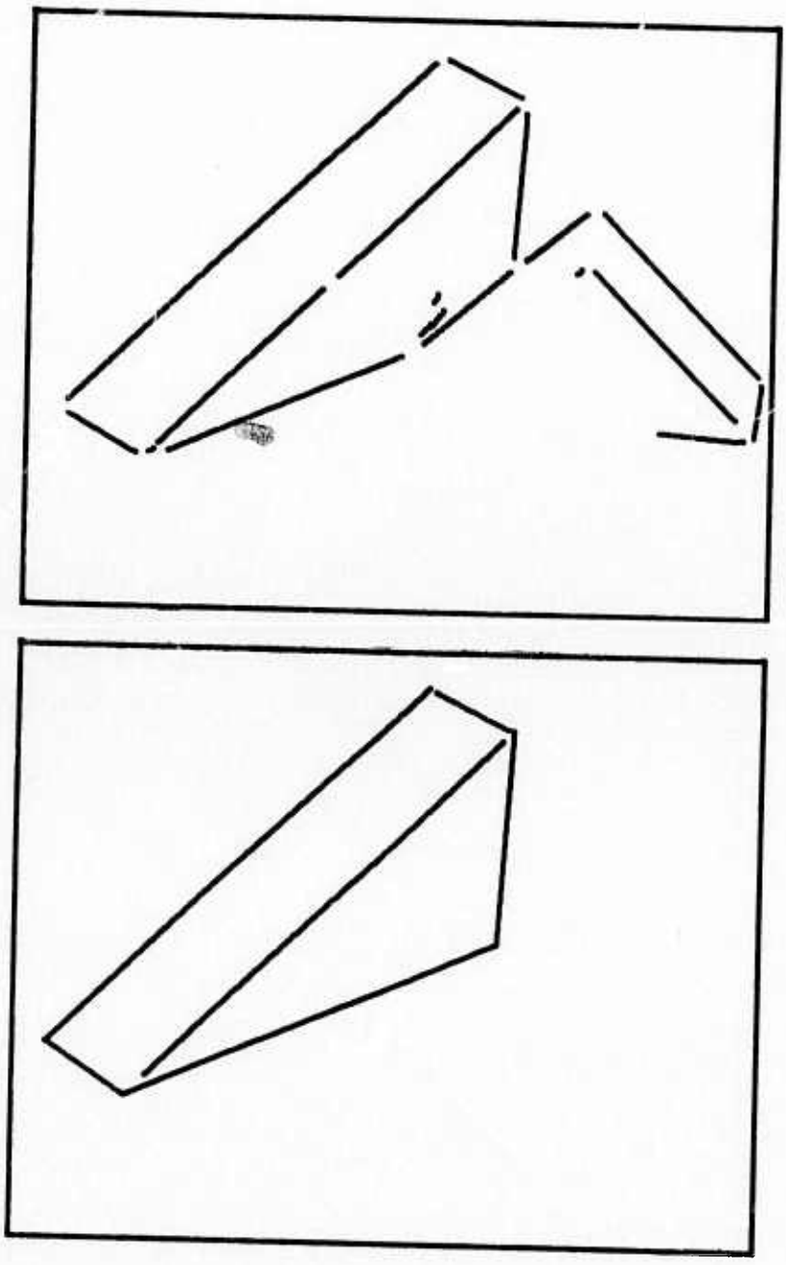


Figure 11.27

SC2: Residual scene - Second object

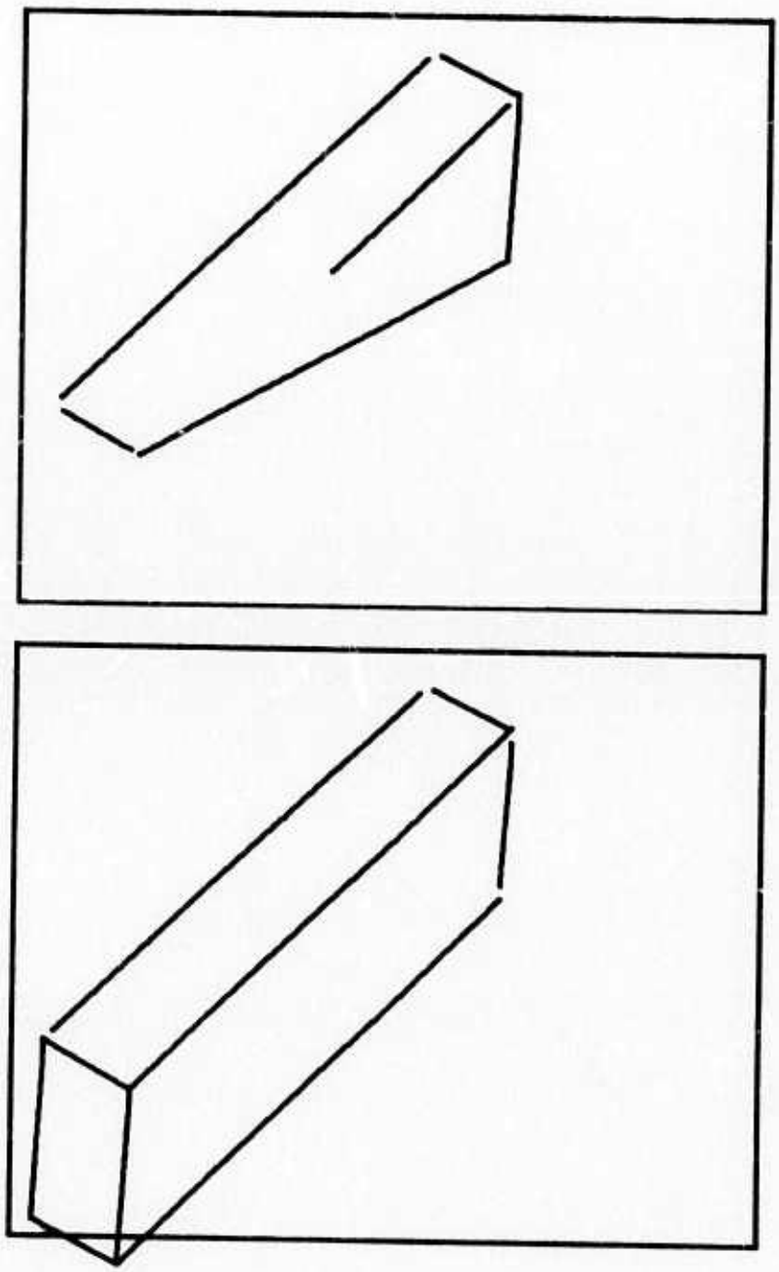


Figure 11.28

SC2: Alternatives for second object - DWEDGE & PAREP

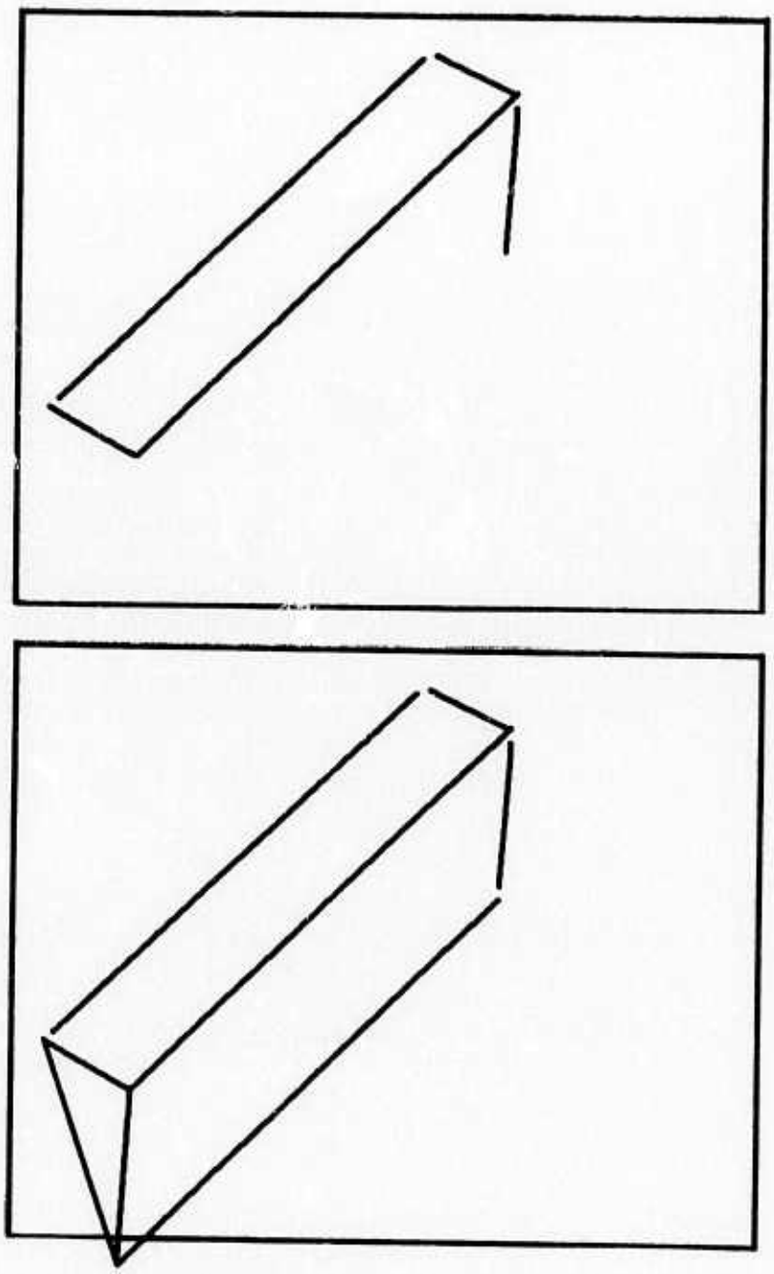


Figure 11.29

SC2: Alternatives for second object - Wedges

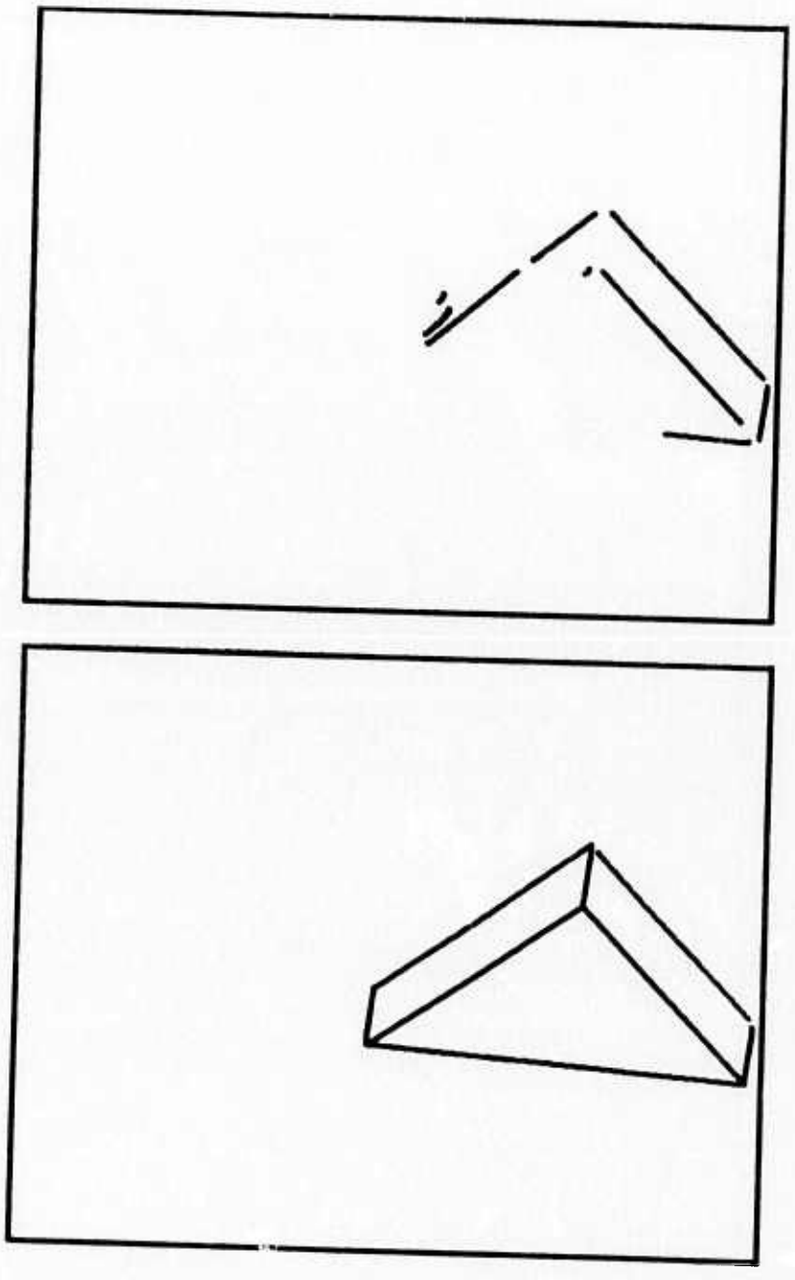


Figure 11.30

SC2: Residual scene - Third object

The final amended scene, and the object superimposition, are given in Figure 11.31. The normal imperfections did not cause much trouble in this scene. They are left as impossible.

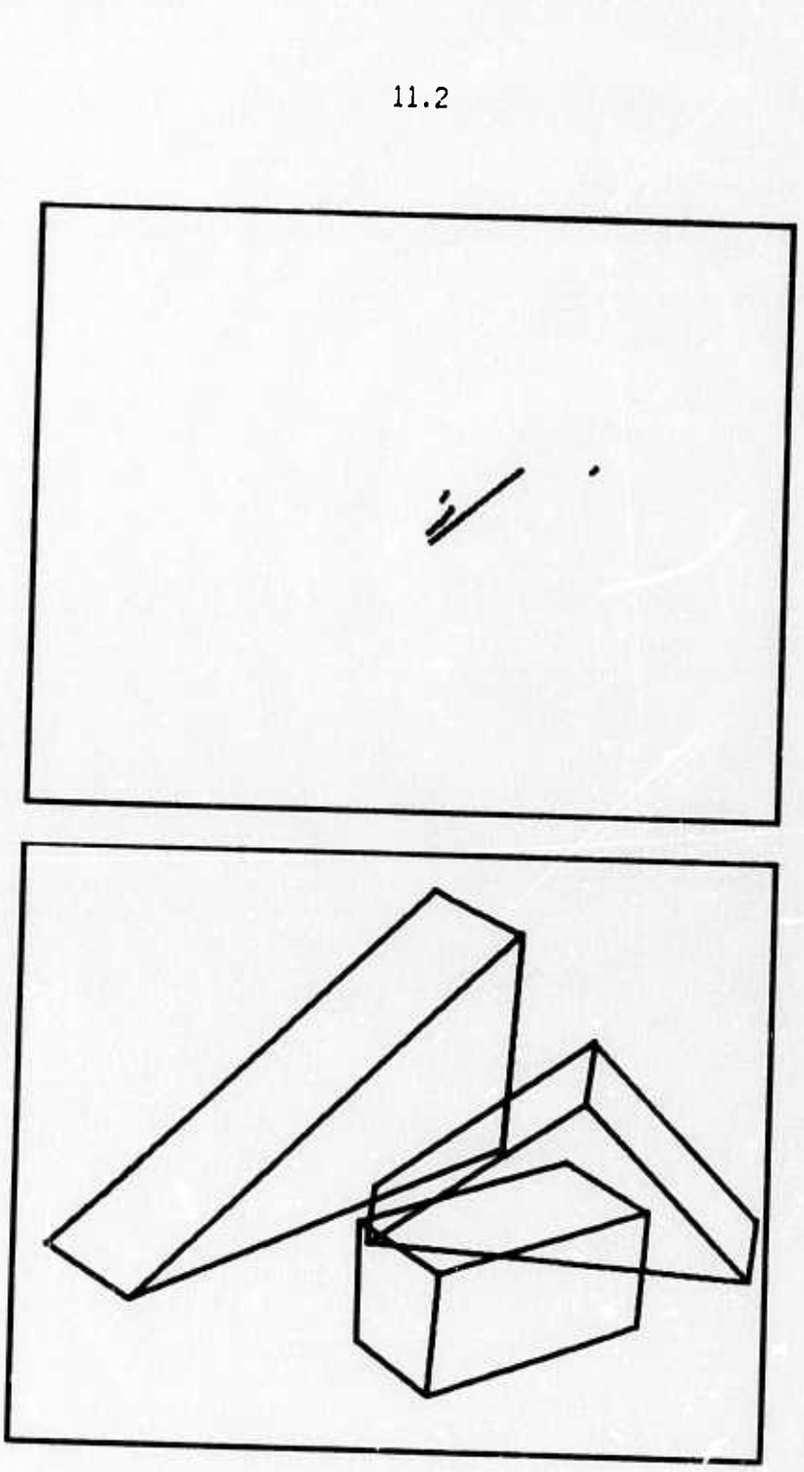


Figure 11.31

SC2: Residual scene - Final interpretation

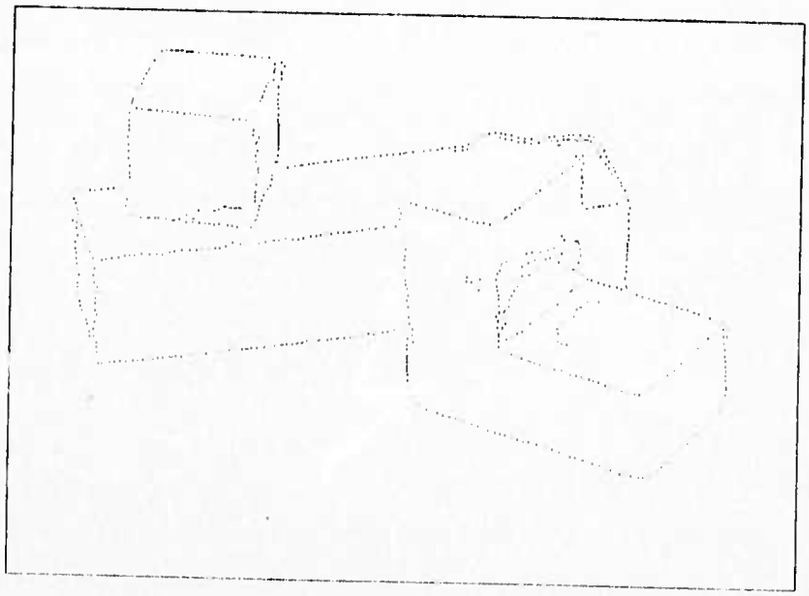
The next example is included because it demonstrates the potential usefulness of features of connection independent elements, such as constellations of parallel lines. Figure 11.32 shows this scene, which contains a concave object, namely an L-beam. So this example also indicates how concave objects may be thought of as consisting of several recognizable parts, which is discussed in Section 12.

Figure 11.33 and Figure 11.34 should need no comments by now.

The case of the L-beam is more interesting. Figure 11.35 demonstrates how the program deals with this situation. It finds a parallelepiped at the bottom, thereby splitting the object into two. It preferred the longer version of that PAREP to the shorter one, due to the perfect outline at bottom left.

Anyhow, in the amended scene (top of Figure 11.36) I can clearly see a parallelepiped. The program could not. The reason is that there is not one good vertex around, which might provide a starting point. Here is where global features, based on vertex independent line constellations, would have been most useful. It is easy to see how, for instance, a parallelity feature might have provided a key into the mapping of this object.

If the top of the L-beam had been found, the latter object would have been neatly and automatically split into two recognizable parts. In general, concave objects would necessitate more special treatment, as discussed in Section 12.



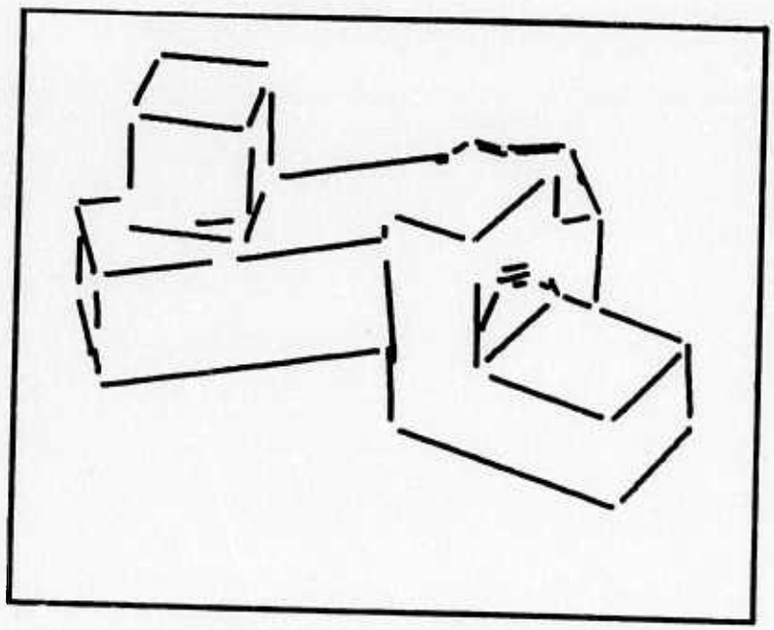


Figure 11.32

SC8: Edge-drawing - Initial lines

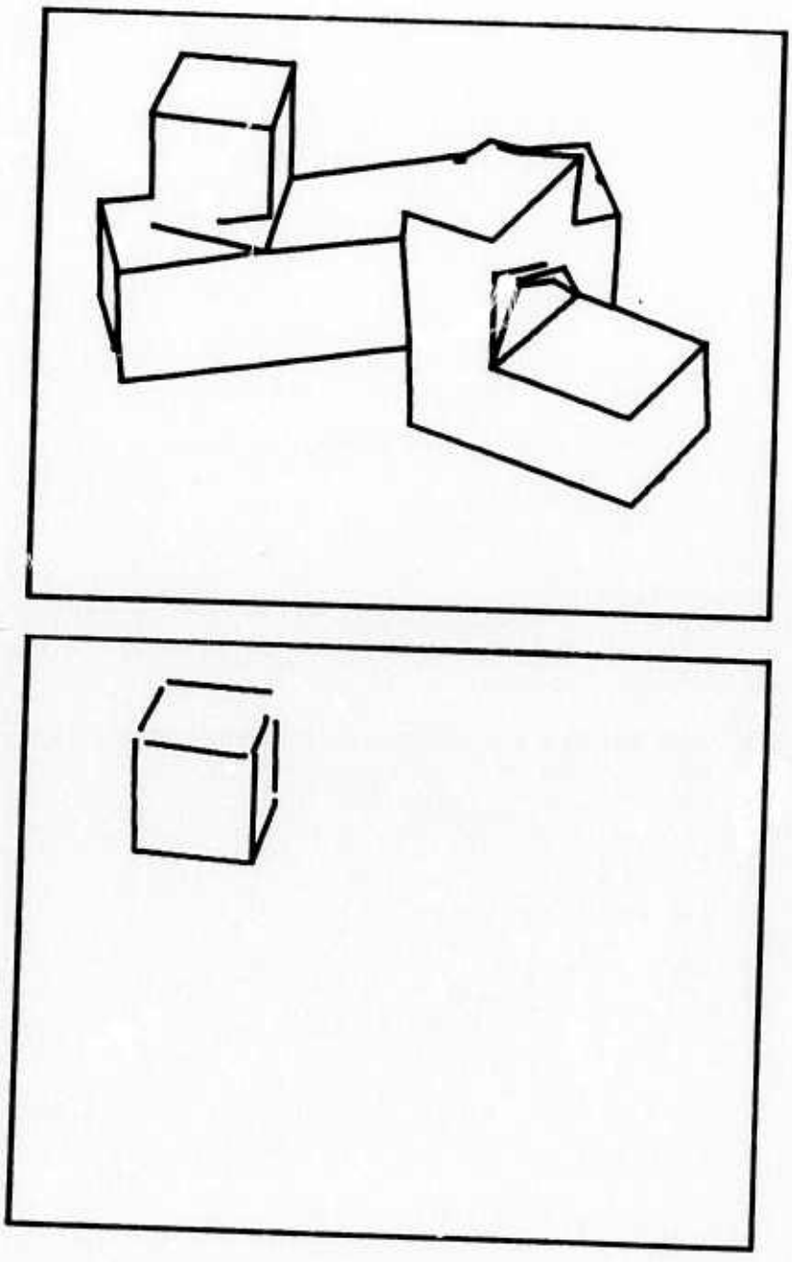


Figure 11.33

SC8: Tentative vertices - First object

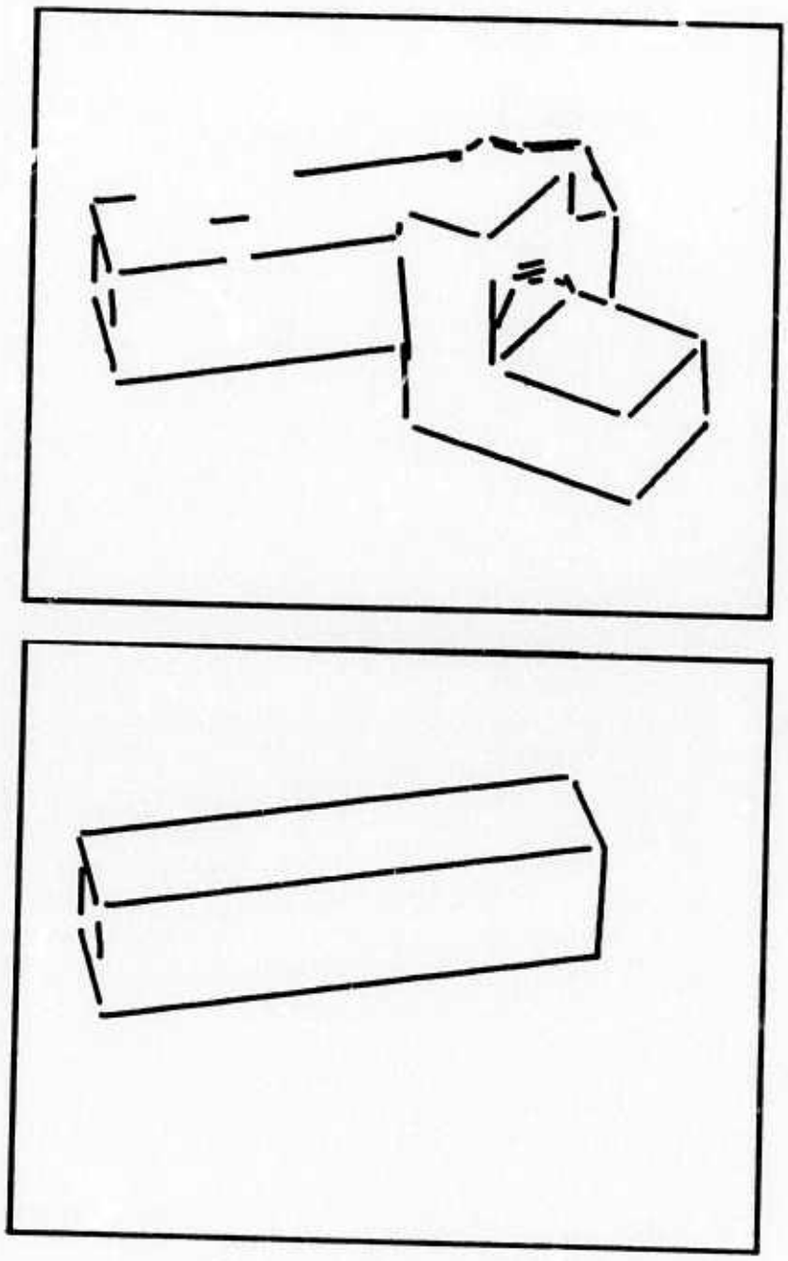


Figure 11.34

SC8: Residual scene - Second object

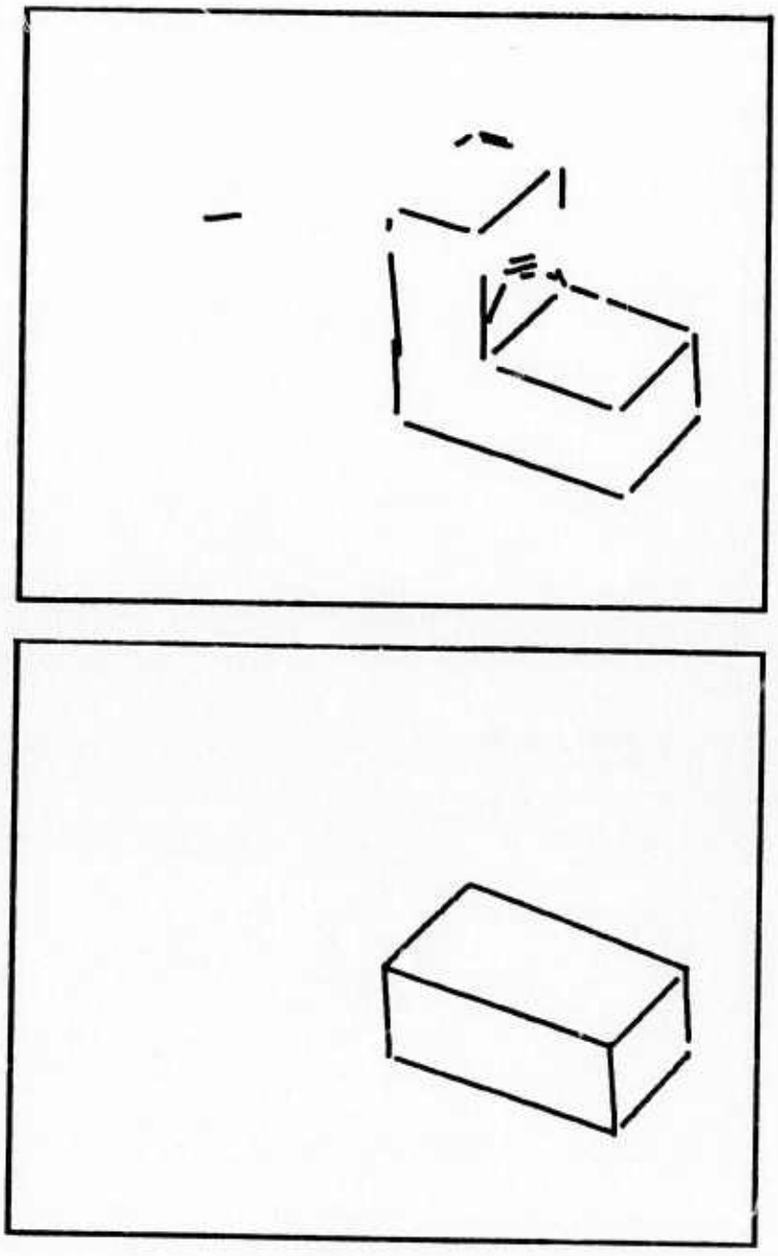


Figure 11.35

SC8: Residual scene - Third object

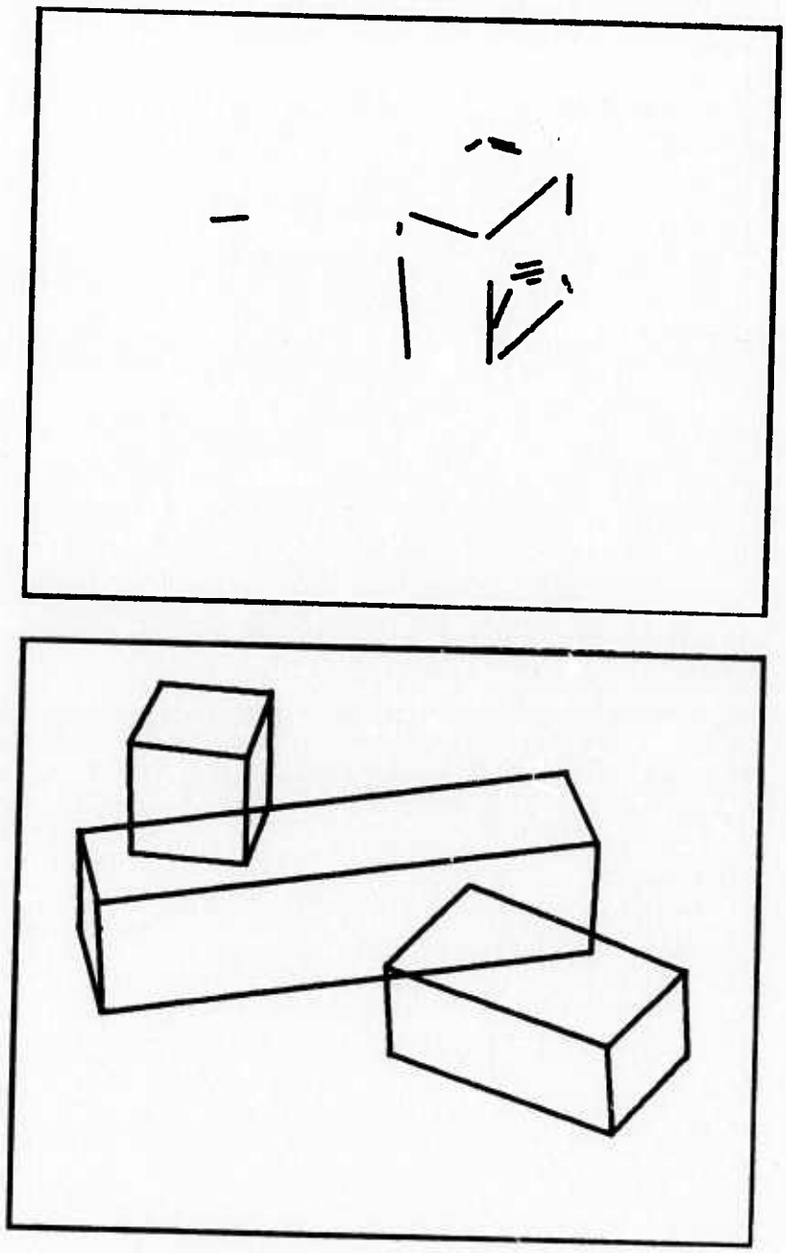


Figure 11.36

SC8: Residual scene - Final interpretation

The following scene - the seventh presented here - is the most complicated one, in terms of the number of objects. It is also difficult due to the small overall scale. Figure 11.37 shows this scene.

In the line-drawing (bottom of Figure 11.38) even I (though a human) do not know exactly what is going on, since there are many lines missing in strategic places, as well as unwanted ones present in other places. It is very noticeable in this example how the parsing strategy of isolating one object at a time (and removing the lines belonging to it) has the effect of cleaning up the picture, thereby facilitating subsequent work.

The first object is extracted without difficulty (Figure 11.39), however with some distortion due to glare effects.

The extraction of the second object (Figure 11.40) does not present anything new and exciting, either.

The case of the parallelepiped resting on the wedges is more interesting. Here the matcher initially finds a much shorter PAREP (I think you can easily see where), which is eventually discarded when further investigation yields the longer (and better) one, shown in Figure 11.41. The bottom right line is not assimilated into this object due to a slight distortion of the front face.

In Figure 11.42 the wedge on the right is found. Not trivial - but we have seen similar examples previously.

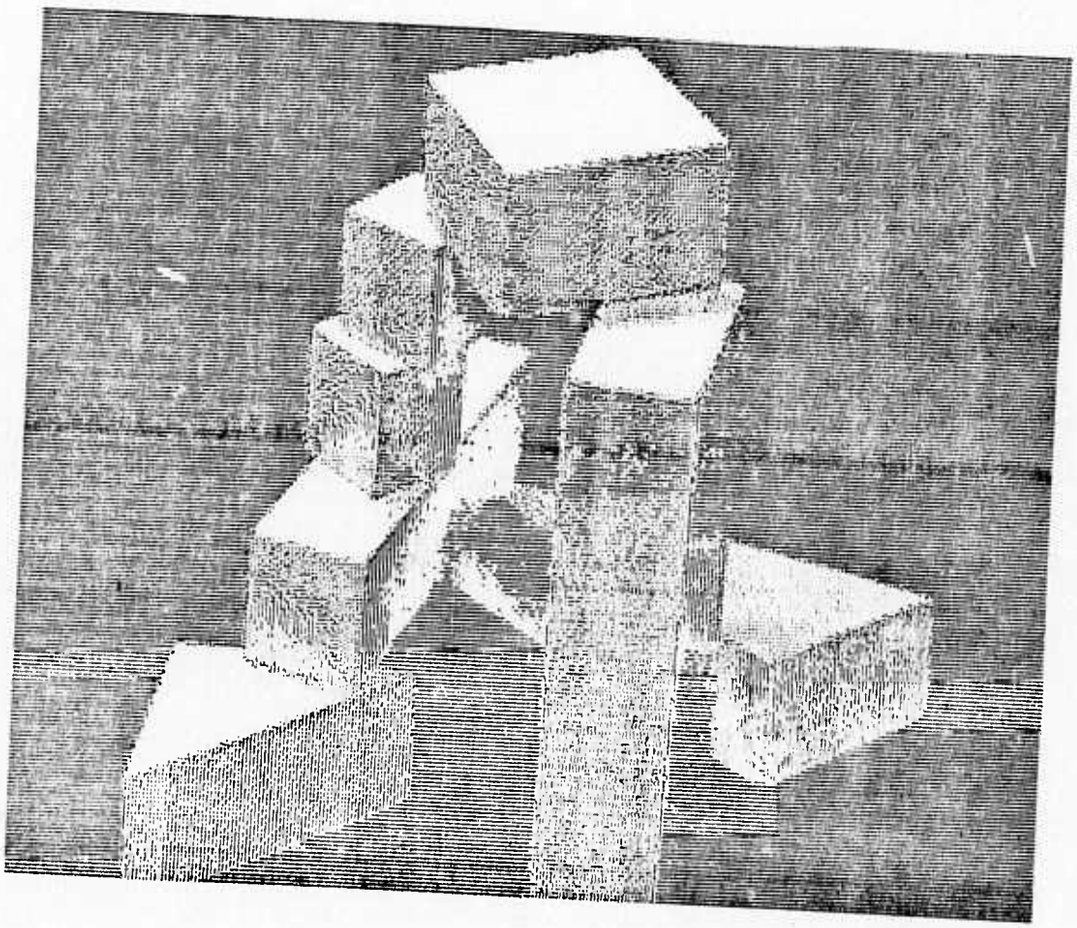
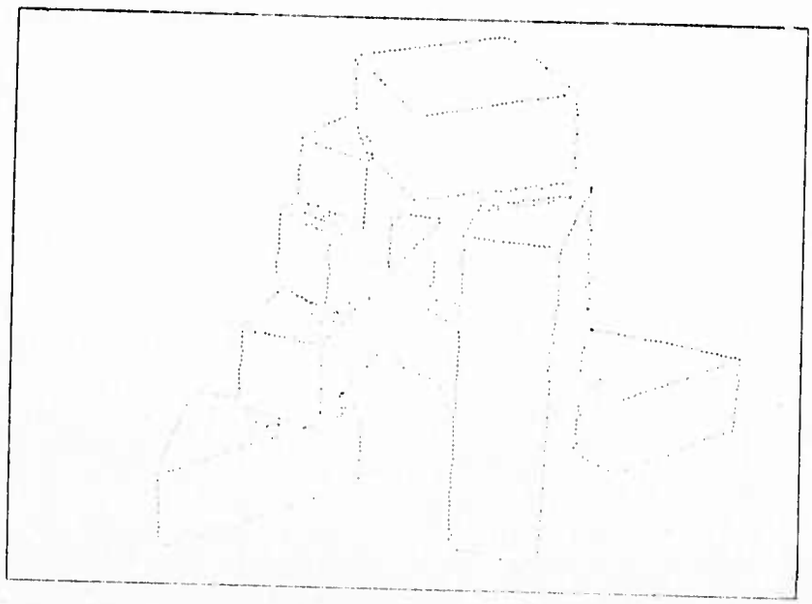


Figure 11.37

SC14: TV-image



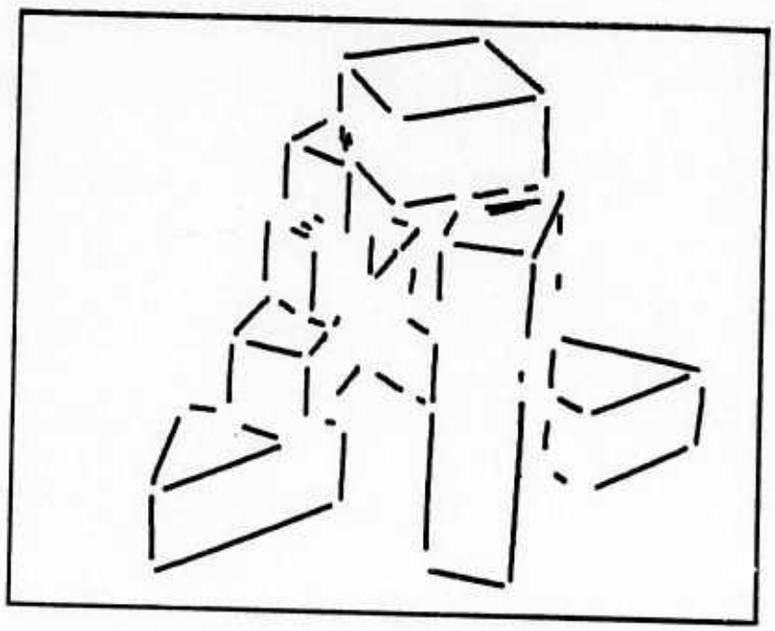


Figure 11.38

SC14: Edge-drawing - Initial lines

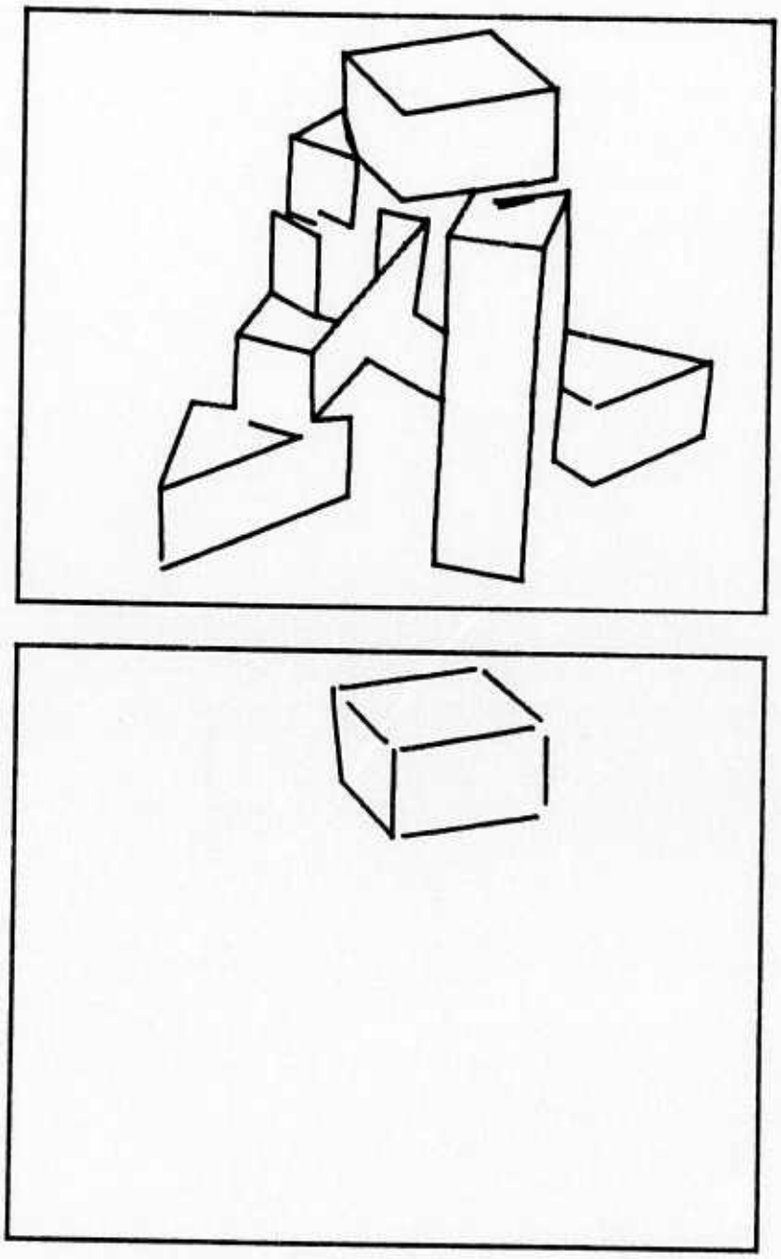


Figure 11.39

SC14: Tentative vertices - First object

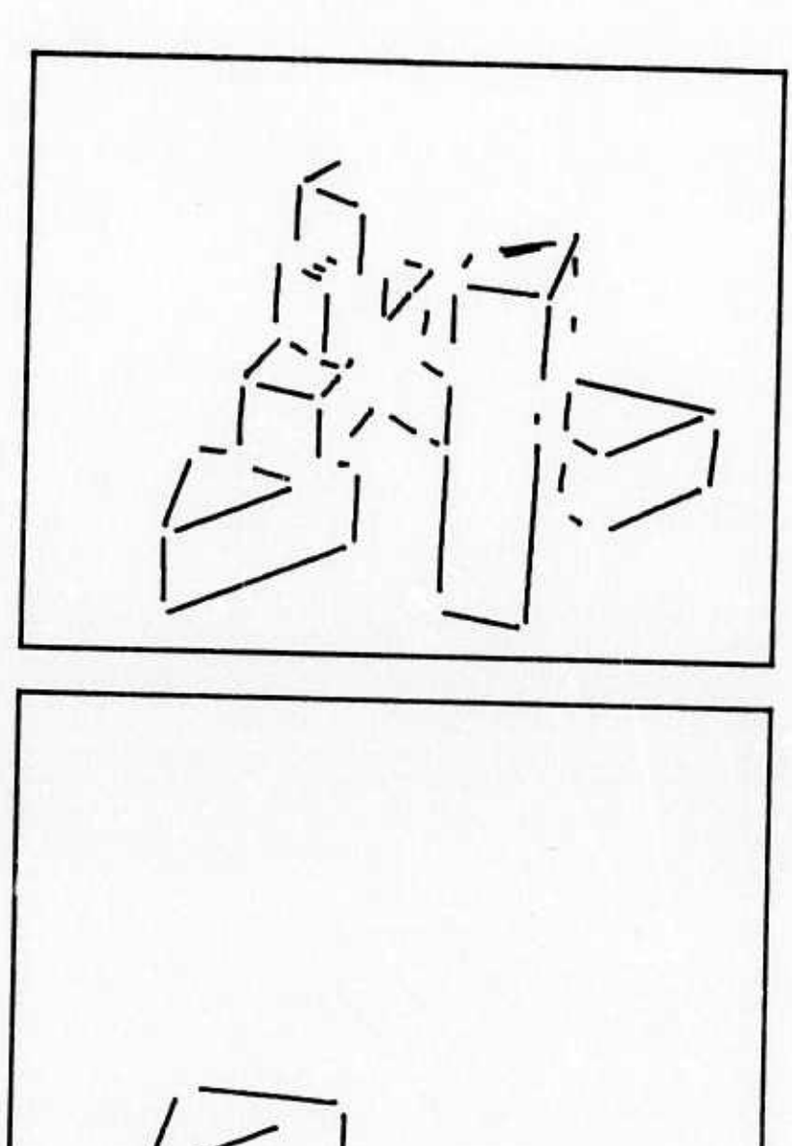


Figure 11.40

SC14: Residual scene - Second object

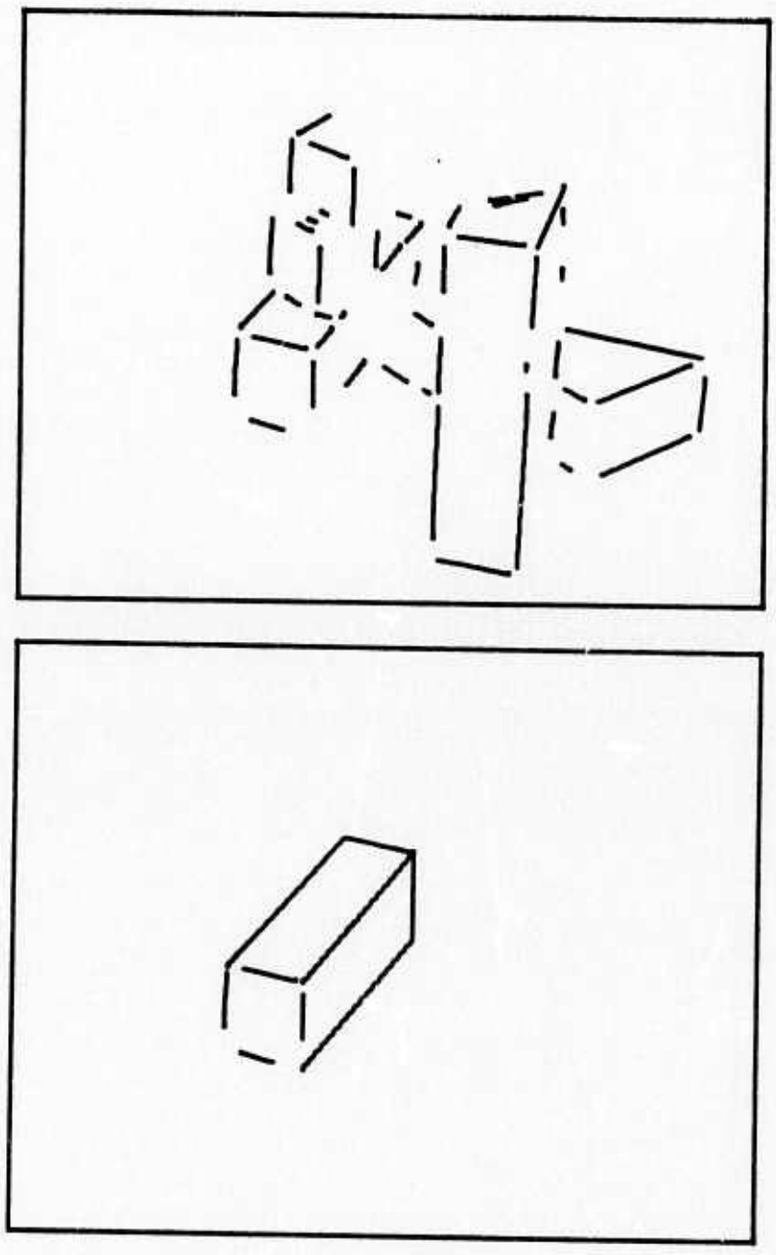


Figure 11.41

SC14: Residual scene - Third object

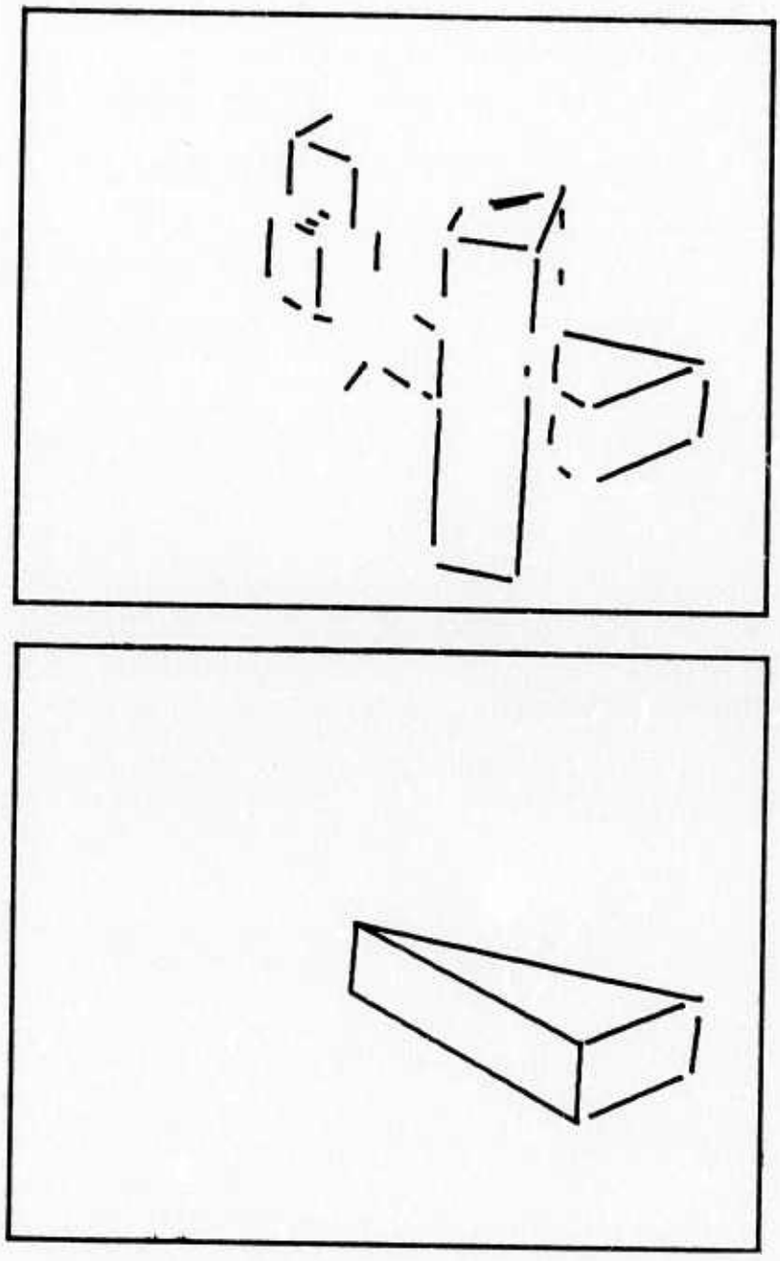


Figure 11.42

SC14: Residual scene - Fourth object

That is the case with the tall, upright parallelepiped as well, which is the next object to be isolated (Figure 11.43).

Finally the program finds two small, partially mapped parallelepipeds, as shown in Figure 11.44 and Figure 11.45. There is no way to tell where they end, and I couldn't have done better myself, on this scene.

The resulting interpretation is presented in Figure 11.46.

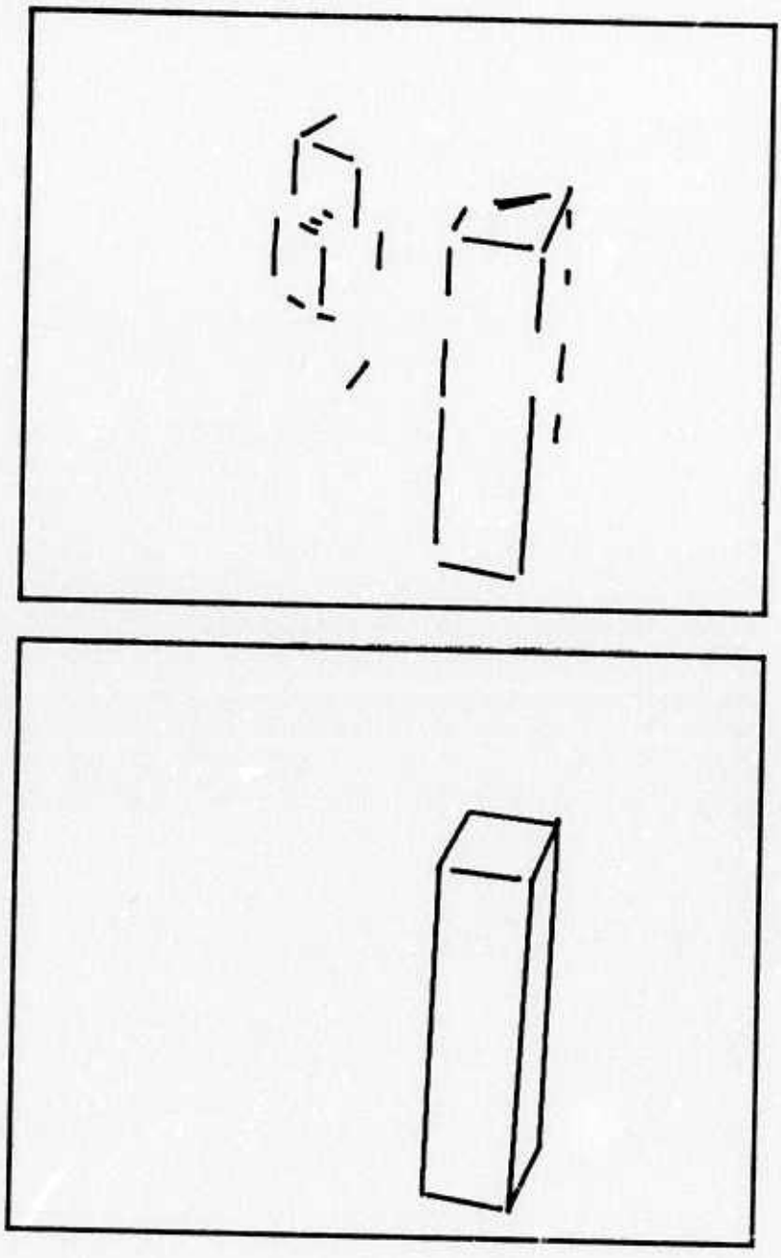


Figure 11.43

SC14: Residual scene - Fifth object

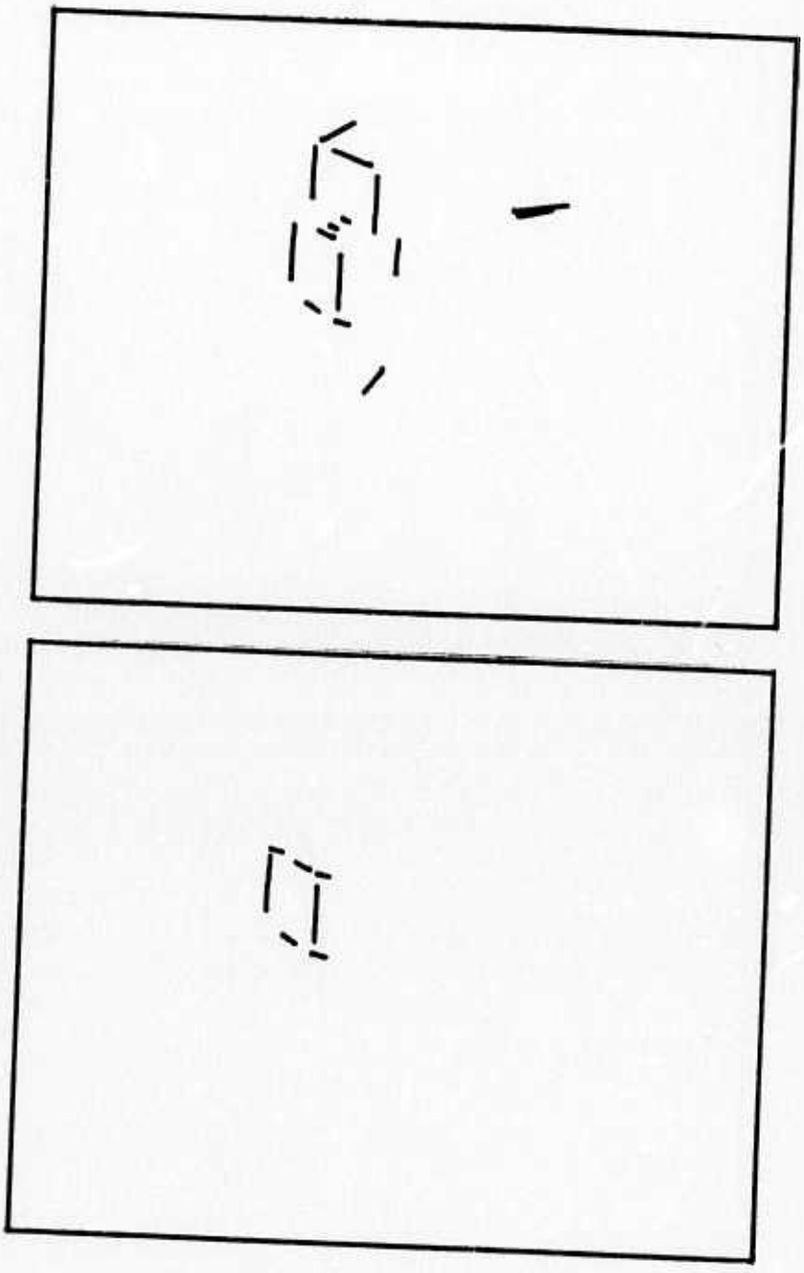


Figure 11.44

SC14: Residual scene - Sixth object

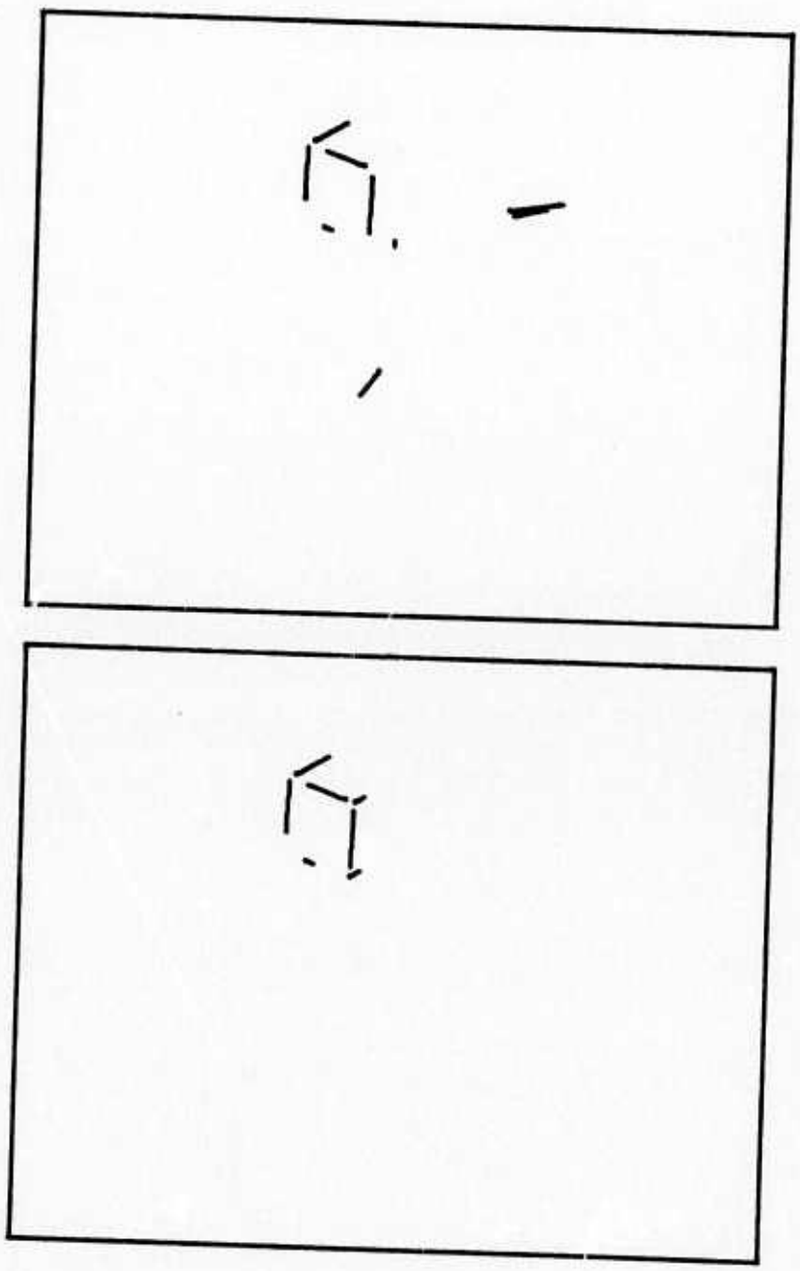


Figure 11.45

SC14: Residual scene - Seventh object

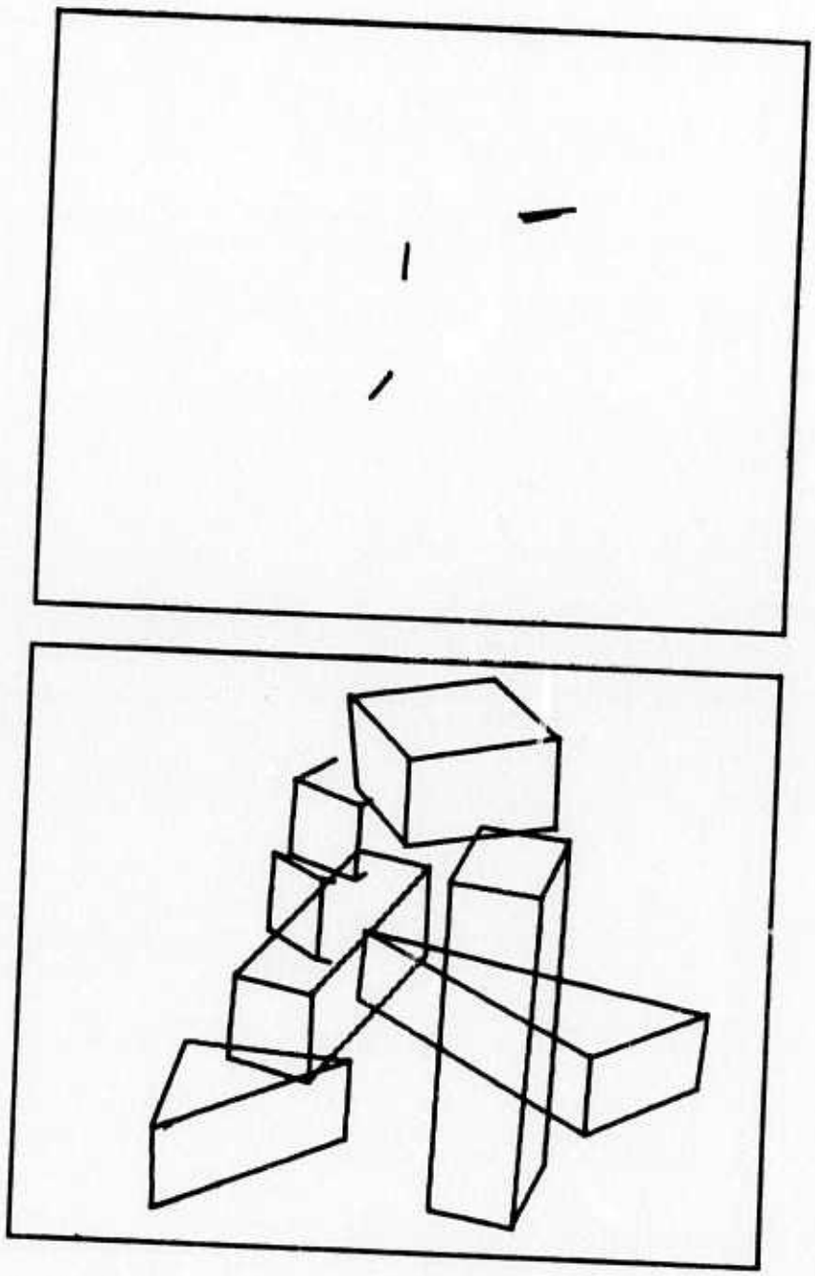


Figure 11.46
SC14: Residual scene - Final interpretation

The last example does not contain as many objects as the previous one, but it is far messier in terms of shadows and glare (Figure 11.47). In fact, one of the objects (the wedge on the left) could not have been recognized without the use of partially similar features as keys.

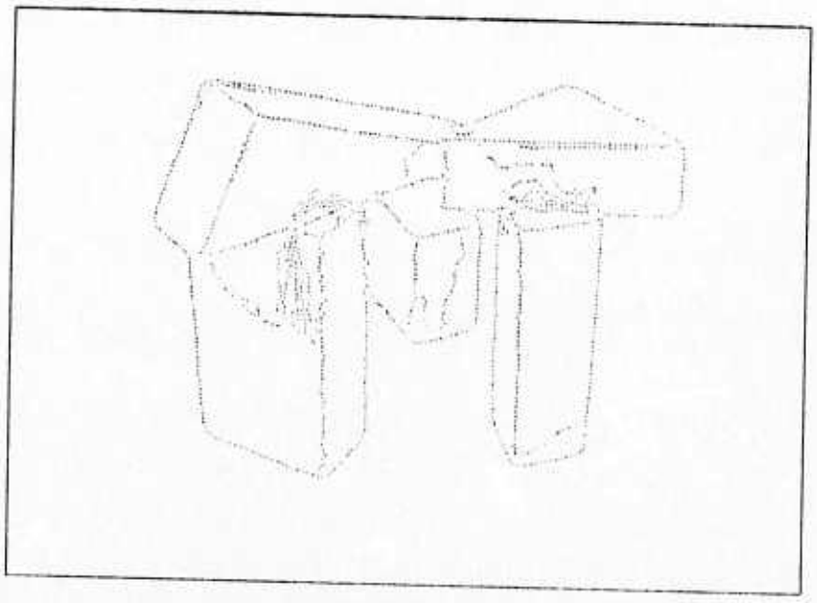
The wedge on the right is the first to go (Figure 11.48). There isn't much to say about that.

The second object (Figure 11.49) is identified somehow (it is hard to see how the lines are linked at the bottom), as a wedge. This seems plausible enough, judging from the evidence at the top. I happen to know that the object was a parallelepiped - but that is beside the point.

Figure 11.50 shows the extraction of the tall, narrow parallelepiped. Due to the way the lines are linked at the top of that object (cf. top of Figure 11.48) the program uses the short line for the left side. The longer neighbour is close enough to be assimilated into the object. The bottom vertex is extrapolated, leaving one line unused. The mapping is good enough for acceptance at this point, and the program exits without investigating further.

Next to go is the big parallelepiped (Figure 11.51). No mean trick - but not much different from things we have seen before.

The isolation of the wedge (Figure 11.52) is more interesting, since that object contains no recognizable features. The heuristic for using



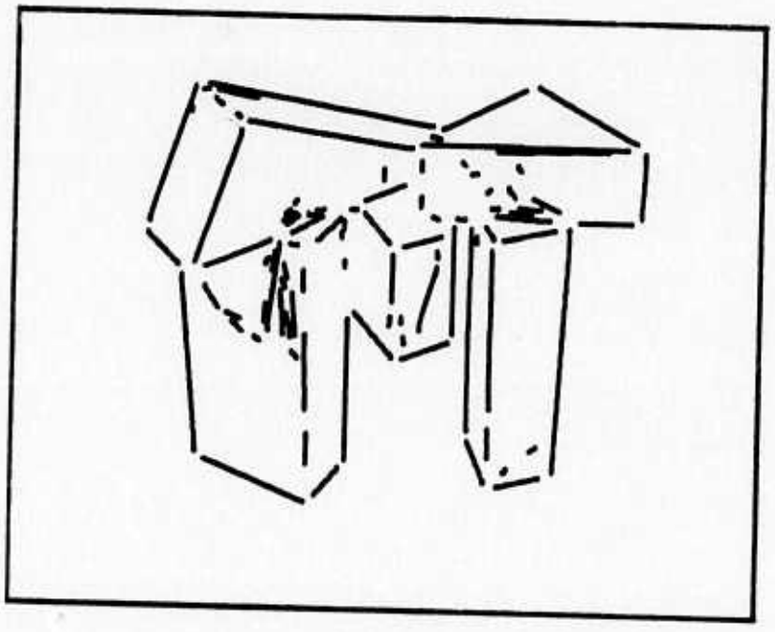


Figure 11.47

SC9: Edge-drawing - Initial lines

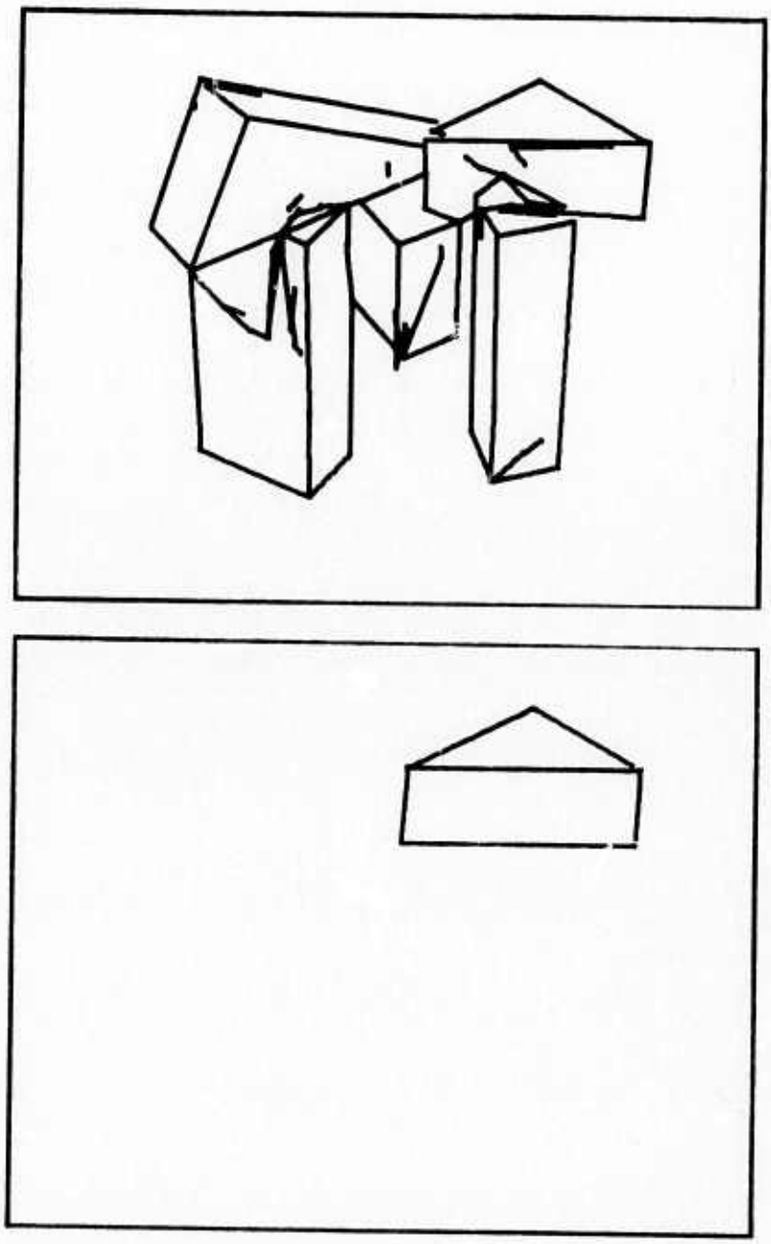


Figure 11.48

SCENE: Tentative vertices - First object

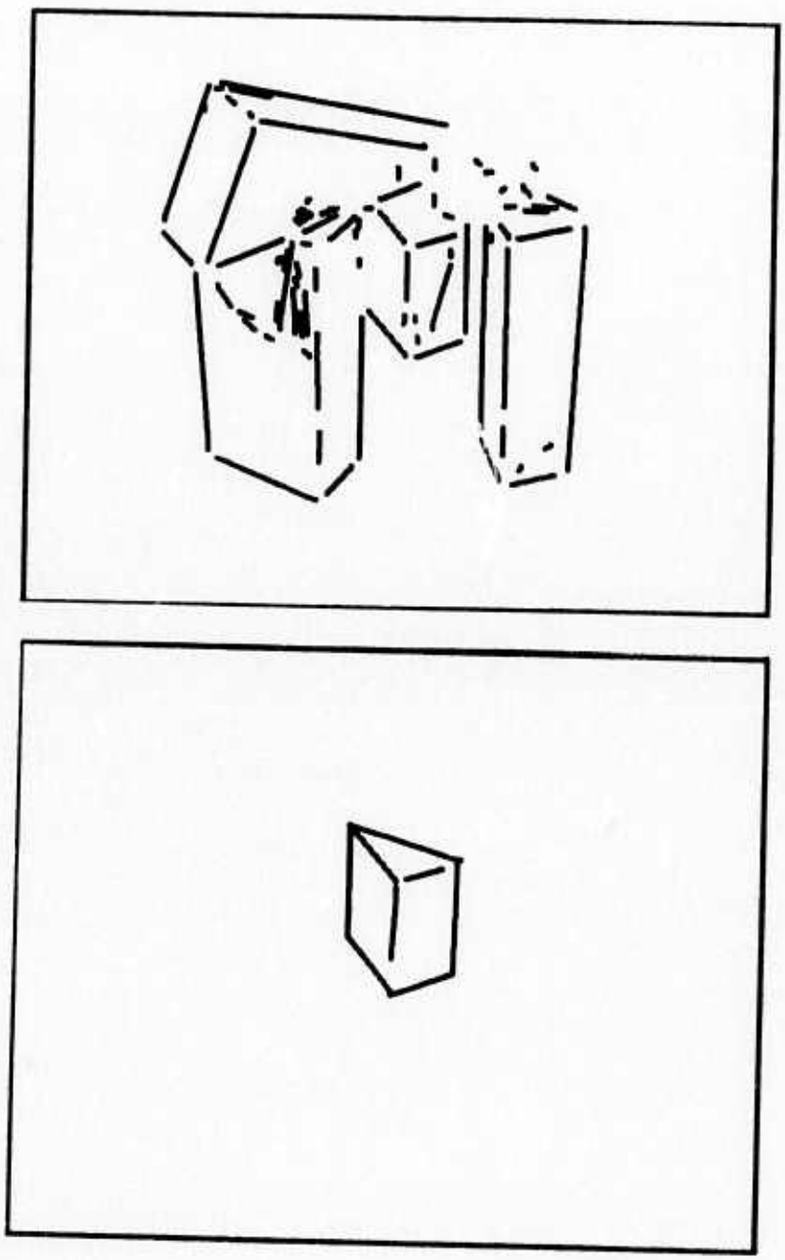


Figure 11.49

SC9: Residual scene - Second object

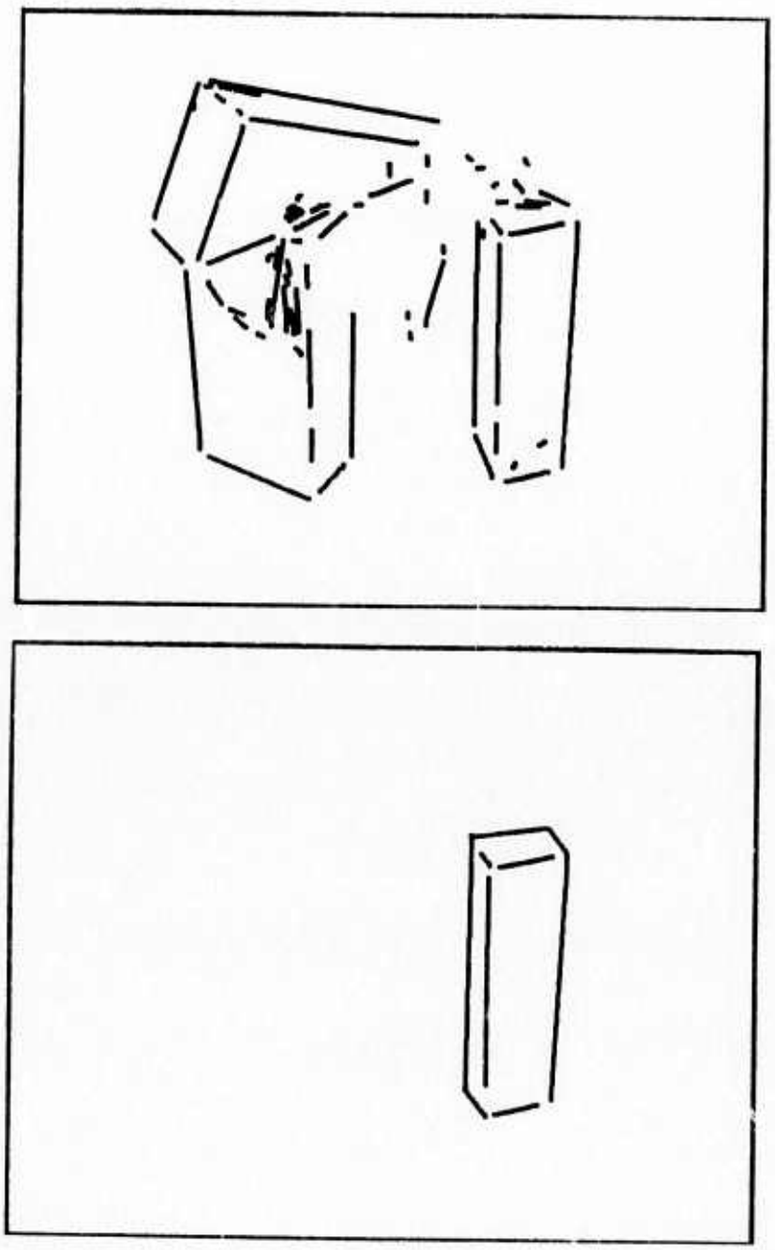


Figure 11.50

SC9: Residual scene - Third object

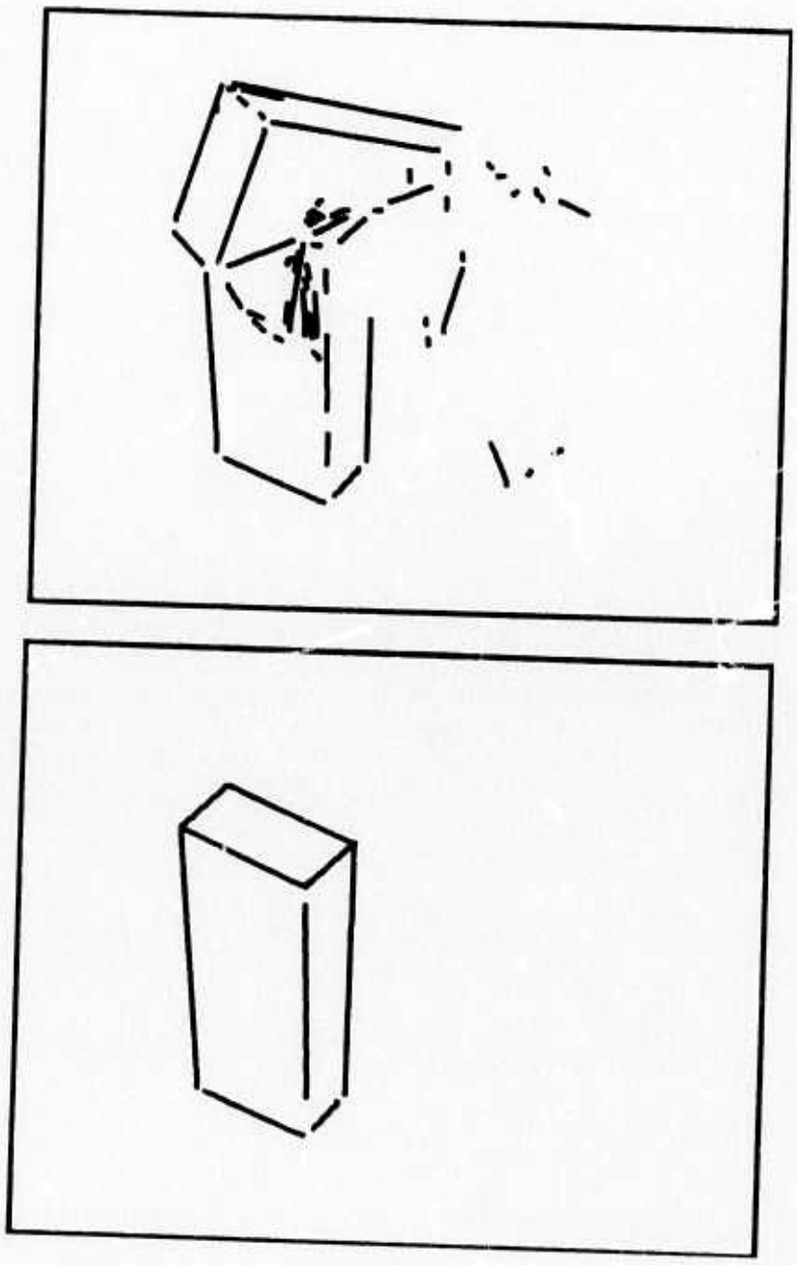


Figure 11.51

SC9: Residual scene - Fourth object

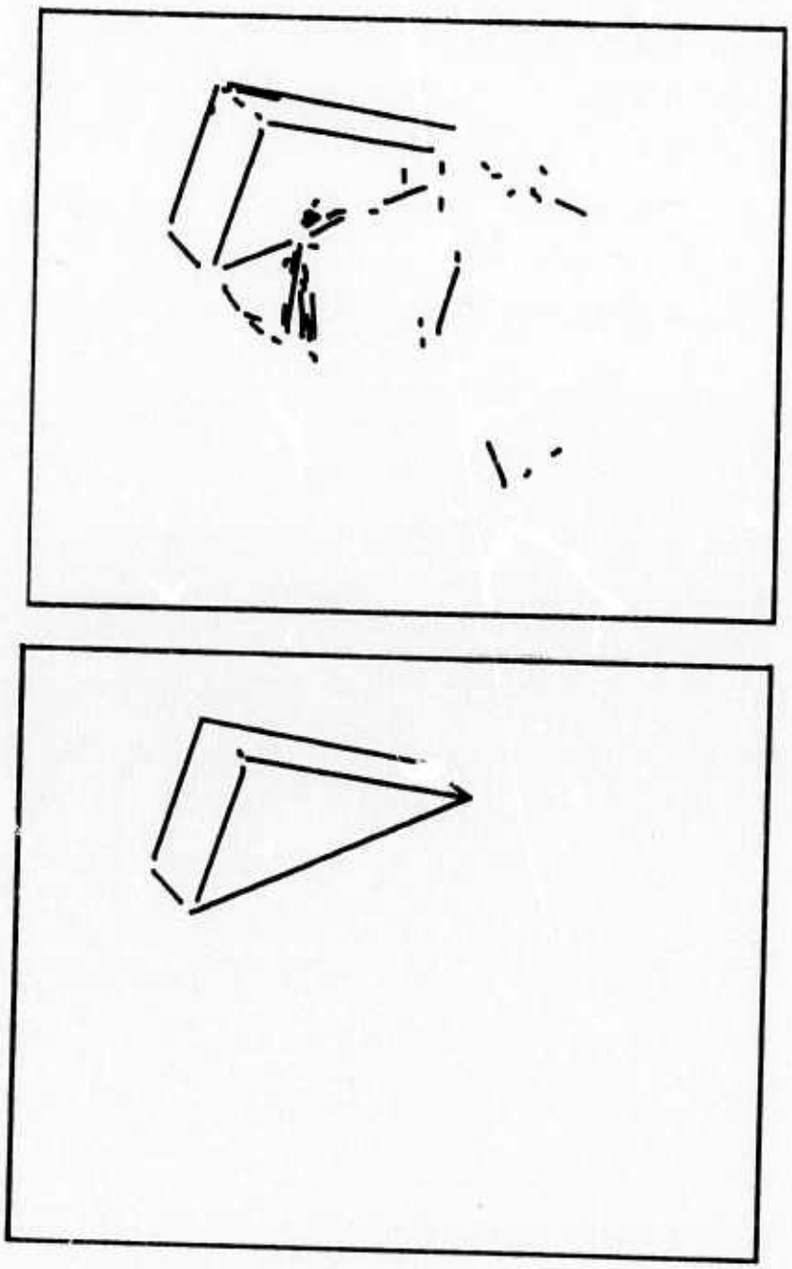


Figure 11.52

SC9: Residual scene - Fifth object

partially similar features as keys is responsible for the success in this case, by determining that it could create recognizable features by disregarding the shadow line at the lower vertex of the wedge. The object can then be extracted without difficulty.

The amended scene (Figure 11.53) is messy enough to provide the parser with one more possible object, which is found in the shadow effect on the front of the central object (cf. Figure 11.47). It finds the parallelogram of a degenerate wedge, the missing two lines of which are assumed, non-directional rays. The present program has no way of knowing its mistake, whereas a complete system (using depth etc.) could better realize the nature of the situation.

Thus the resulting scene interpretation in Figure 11.54 contains that non-object as well, which is basically all right from the standpoint of the present system. Left in the residual scene are the rest of the shadow- and glare lines, a messy lot which did not mislead the program.

This concludes the presentation of examples of system performance.

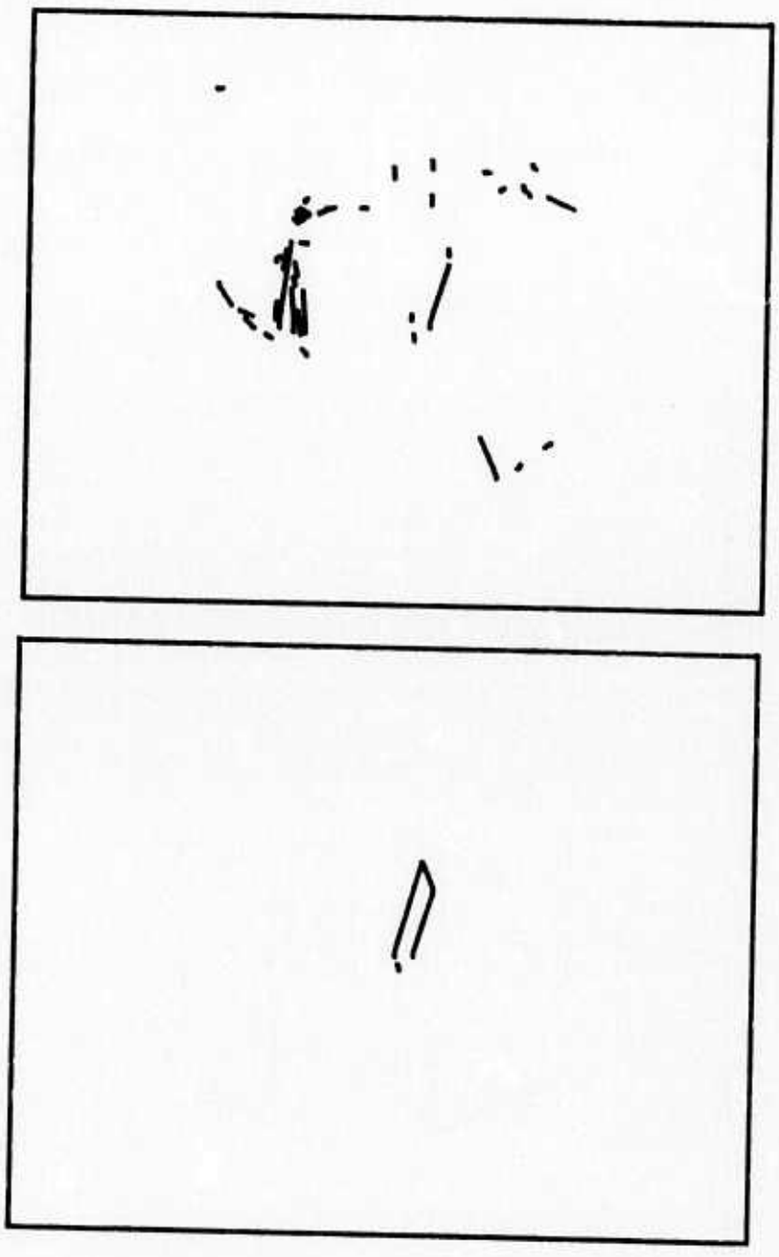


Figure 11.53

SC9: Residual scene - Sixth object

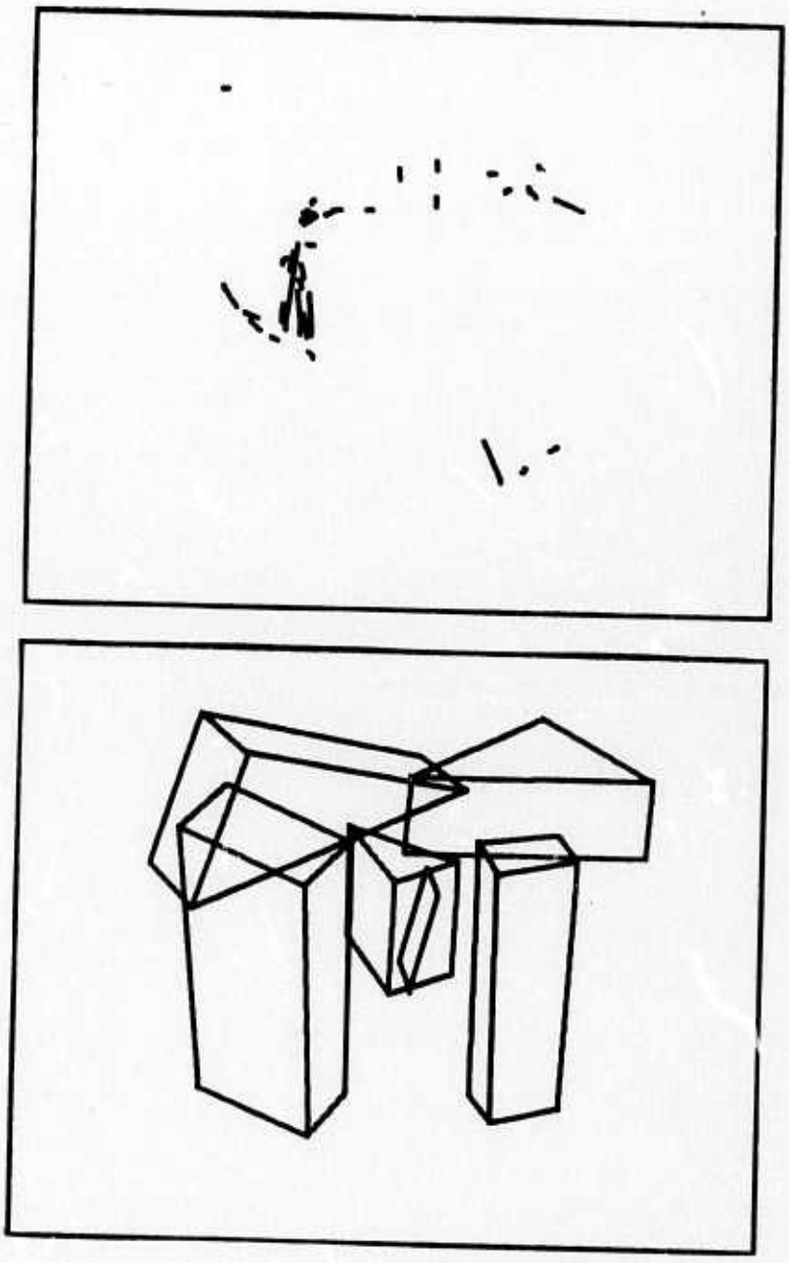


Figure 11.54

SC9: Residual scene - Final interpretation

11.3 DISCUSSION OF SYSTEM PERFORMANCE

Actually, not much of a discussion should be needed here, since the examples are thoroughly commented.

On the scenes tested so far, the present limited system has performed as well as could be hoped. It is able to parse scenes of many objects, in the presence of a good deal of disturbance. In fact the utilization of partially similar features as keys makes it possible to correctly identify objects with only one good vertex, provided one of the lines to that vertex is unbroken.

Sometimes partially mapped objects are classified somewhat haphazardly, but their classifications are not intended as final. A complete system could further process them, since the details of their mappings are remembered.

The CPU-times for the examples given above range from 1 to 6 minutes, typically staying around 2. The time is proportional to the square of the number of lines, and roughly to the squares of the numbers of prototypes and partially mapped objects (since full matches cause quick exits). The dependence of computing time on the square of the complexity of the picture is a weakness inherent in a system based on models. It might be alleviated by the use of more extensive feature schemes, as indicated in Section 12 (future work).

Little effort has been made in the direction of speeding up the program.

It could fairly easily be modified to run substantially faster, by programming frequently used routines directly in assembly language, rather than in the Algol subset of SAIL (approximately Algol, plus associative features) [Swinehart & Sproull 1971].

Let us turn now to a discussion of the possible directions in which work on the present system might proceed.

12.0 FUTURE POSSIBILITIES

The most immediate areas of possible future work concern extensions of the present 2D system. More general aspects involve the use of concepts of 3D in the development of a complete vision system.

12.1 EXTENSIONS OF THE FEATURE CONCEPTS

The feature concept, as implemented here, has the weakness of demanding connectivity. It is certainly possible - and might even be worth-while - to extend it to certain constellations of unconnected lines, such as parallel pairs or triplets, and relationships of such. The example of the L-beam demonstrates the potential advantages of connectivity independent features.

Such features would be useful as guides for a matching supervisor program, in that they could provide an extended context in some cases. Of course they might also be helpful in guiding the process of initial mapping.

Two immediate, more specific possibilities for extensions of the feature concept are the following.

It often happens that lines are broken up by intervening objects, or for other reasons. Most likely the line-features of the parts will not be

recognizable. A future possibility here is to detect such cases and temporarily insert compound lines, under the provision that the created LF:s are recognizable. The relative messiness of such a scheme motivates a "wait-and-see" attitude here.

The second scheme I had in mind concerns the introduction of the SF, which here stands for "super-feature" (not San Francisco). In a super-feature, which may be an extension of the LF concept, we would provide (partial) information regarding the LF-designations of all the participating lines in the feature. Such features would then reference, directly or indirectly, all lines in simple prototypes, providing wider contexts and extremely strong clues to mappings. Of course, line-drawings of scenes are usually messy enough that complete SF:s would be rare. We would almost always have to use partial ones, which is all right.

In any case, SF:s could provide initial mappings (keys), based on much broader contexts than do the present LF:s and CF:s. In fact, SF:s could conceivably guide the parsing process, providing the order in which to explore the keys. I see no use for SF:s in the matching process itself.

12.2 RECOURSE TO INITIAL DATA

The present parsing program decides between different mappings on the basis of the lines present in the initial line-drawing. In the context of the Stanford Hand-Eye Project it is fully possible to base such judgments on data from the original TV-image, since that system includes a statistically based line-verifier that operates on the digitized TV-raster [Tenenbaum 1970].

While on the subject of having recourse to the TV-image itself, I should mention the possibility of a "closer-look" strategy. This would apply in areas of insufficient or confusing information, and would entail sensitivity accomodation as well as (and perhaps especially) changing the lens into one of greater magnification. Details regarding the technicalities of related subjects may be found in [Sobel 1970].

12.3 EXTENDED CONTEXTS AND 3D

The most interesting possibilities arise in the extended context of 3D.

In a full-fledged three-dimensionally based system, with access to depth-information, the basic prototypes would be given by (fictitious) coordinates in space, and the final scene interpretation would be based on spatial considerations, support-theorems, etc.

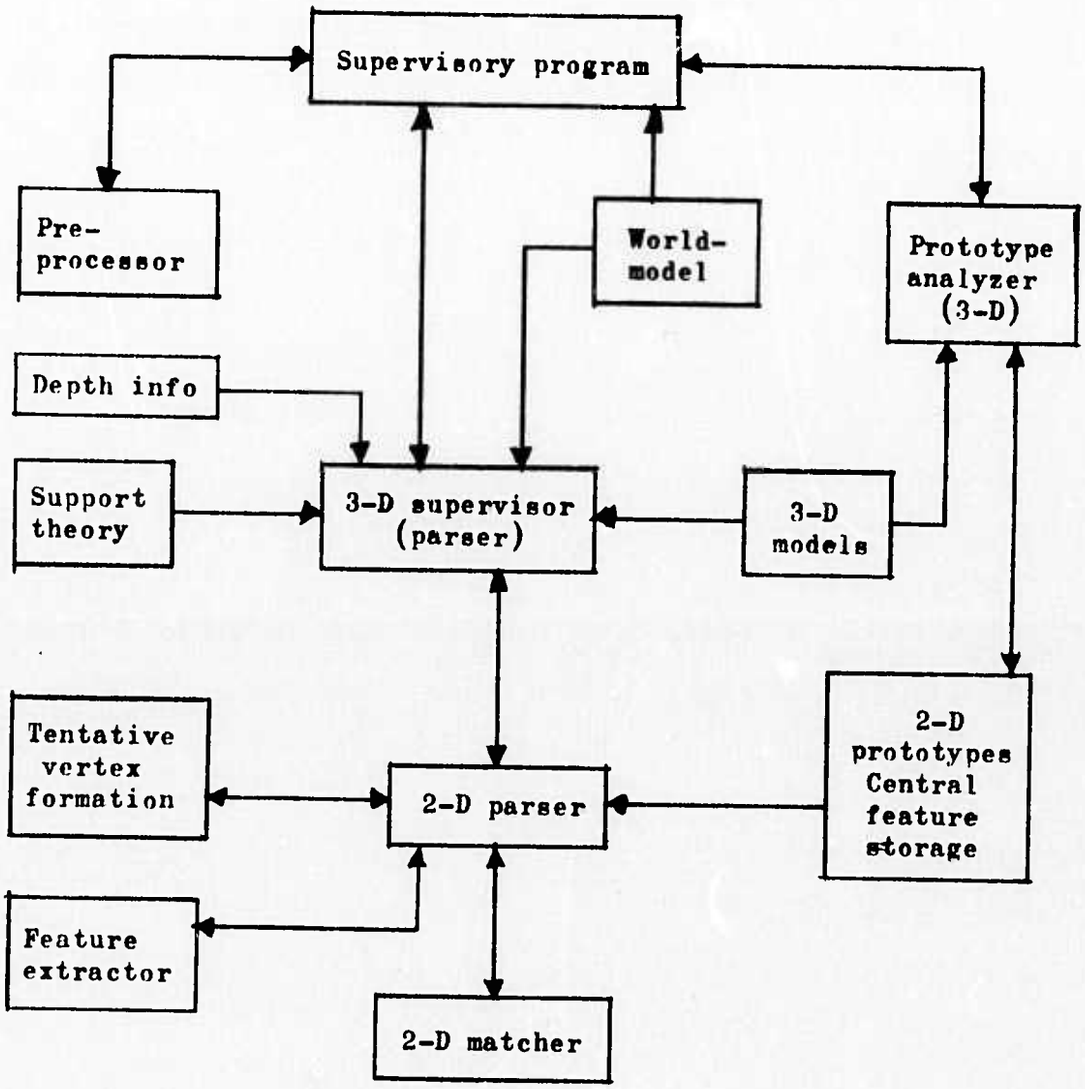


Figure 12.1

Possible information flow in a 3D system

Figure 12.1 presents a diagram suggesting the flow of information in such a system.

The feature scheme, and the 2D prototype matching scheme, could be much the same as now. The 2D prototypes would be generated automatically from models in 3-space. The prototype analyzer would generate all different views of an object, checking each one against existing 2D prototypes, and updating that memory structure whenever necessary. Handy programs for the creation and manipulation of 3D scene-representations exist already [Baumgart 1973], and those should prove most useful in such contexts.

The prototype matching would proceed more or less as it does at present, but the decisions of acceptance and interpretation would now, at least in doubtful cases, be the responsibility of the 3D parsing supervisor, with judgments based on information and theory not available to the present parser, as well as on the specific details of the current world-model, which describes what the environment is expected to be like and what kinds of objects make up the world.

Depth-information may of course be obtained directly, using the laser. Another alternative - well suited to the present system - would be to use stereo correlation, i.e. work in parallel on two different views, separated by an adequate angle (from the point of view of depth-separation).

Since the 2D parser isolates one object at a time ("best first",

basically), the task of identifying and correlating objects between the two views should not turn out to be excessively hard.

12.4 EXTENSION TO GENERAL PLANAR FACED OBJECTS

Roberts (cf. Subsection 3.1) introduced the idea of representing non-convex bodies as composed of two or more convex parts. This certainly seems like a very sound approach, especially in a model-based system, where self-occlusions would otherwise create great difficulties and vastly increase the required numbers of two-dimensional prototypes.

What is needed, then, is a method of describing the junctions of convex objects into more complex ones, so that the parser, having found the parts, may infer the whole (in some representation). The representation of a concave body as a collection of convex parts is at best a highly ambiguous undertaking, which requires rigorous conventions on the part of the prototype analyzer, and a great deal of flexibility on the part of the parser.

It would seem that any meaningful extension to non-convex objects would have to take place in the context of 3D, and would probably require verification loops accessing the TV-image, since it might otherwise be hard to determine whether we are looking at one object adjacent to another, or just at one single, more complex body.

13.0 CONCLUSIONS

A system for intermediate-level computer vision has been developed, which utilizes global information, in the form of two-dimensional models, in interpreting an image as a representation of a three-dimensional scene. The world is assumed limited to planar faced, convex objects.

System performance seems most satisfactory. For scenes of regularly shaped objects, such as our dear old parallelepipeds and wedges, the present system shows good discriminatory power, even under adverse conditions, as in the presence of disturbances like shadows, glare, and missing lines.

The system presented here was created with the extended context of three-dimensional interpretations in mind, and it should prove quite readily adaptable for use in a complete vision system.

14.0 APPENDIX

It was originally my intention to include the mathematics of least-square line-fit and vertex merging by a weighted-least-square method here, as well as the edge-sorting algorithm, collinearity criteria, etc.

However, this paper is long enough as it is, and I don't want to burden it with extra details, unless unusual or otherwise interesting, which to my mind precludes the above-mentioned. I shall be content to give some account of the basic data-structure of the present system.

14.1 THE GENERAL DATA-STRUCTURE

This presentation is intended to provide some general principles rather than implementation details. It will not deal with the data-structures pertinent to features, prototypes, or mappings, since those were discussed in their proper contexts.

Some of the important considerations behind the design of the data-structure were:

Easy random access

List structure for context

An absolute minimum of shuffling

Ability to expand if needed

Preceding page blank

It was designed some time before I developed the present feature- and prototype schemes. It is of a general-purpose character (within its frame-work), and has proven efficient and fairly easy to work with.

The scene-data is classified into three basic groups of pertinence, namely lines, line-ends, and vertices.

The information pertaining to lines is of a more or less physical nature, such as coordinates, coefficients in equation, angular argument, basis in edge-data, etc. A very important item associated with each line is its LCREDE (Line CREation and DEletion) value, to which I shall return below.

Associated with each line is also the linkage of its ends. Those are referred to as SV:s (simple vertices), and they figure mainly in the context of the list structure providing vertex linkages. Thus, for each SV, we have a pointer to its orbital successor line(-end), and the angle to that line, ccw. around the vertex in question.

Normal vertices, where several lines come together, are called CV:s (compound vertices), and each SV also has a pointer to the CV (if any), of which it is a member. Risking confusion, I hesitate to add that CV:s may be single, as well. With each CV is associated a pointer to one of the SV:s in its ring, and also physical coordinates, which are obtained through a weighted-least-square method I developed (which, incidentally, can be used to obtain perspective vanishing points, as well), and which minimizes the squares of the distances from the lines (to the point), weighted by the square roots of the line-lengths.

To sum up, SV:s are line-ends, and CV:s are vertices, and those structures are completely separate. The linkages define the interpretation of the scene-representation.

Therefore, each element, be it a line, an SV, or a CV, has a fixed amount of storage associated with it, which makes its components directly addressable, besides in some cases being members of list structures. Furthermore, deleted elements are linked into free-storage lists, so that no shuffling is needed except when core has to be expanded. This happens when the information content (or the messiness) of a scene exceeds expected bounds.

14.2 THE SUBCONSCIOUS

The above-mentioned LCREDE defines the status of a line, i.e. whether it is part of free storage (no line at all), inactivated, or currently active. It also contains a short, two-level, memory of recent states.

Actually, the top of LCREDE only defines status in relation to two global variables which define the current range of the conscious. By changing those global values, we may forget parts of the scene, and bring other parts to the surface.

As an added possibility for diversity (confusion), we may have vertex connections temporary or permanent, as defined by the signs of the SV-

pointers (orbit-pointers). This has not yet been utilized in the present system, where all links are temporarily permanent.
(I beg your pardon..?)

There is a vast library of subroutines that perform various exotic actions on the scene-representation, and those (when called properly) can be instructed to work only in the subconscious, or the conscious, or the temporarily subconscious, or ... For the present, most routines are instructed to work in the temporarily permanent conscious, that is with active lines only.

The two globals defining the current range of the conscious are manipulated by the parser, the matcher, and various other programs, in the processing of a scene. Since each LCREDE is a short memory stack, lines may be temporarily forgotten (by pushing down), or conveniently recalled (by popping the stack). This possibility is used extensively in the matcher, who is very busy replacing lines or line-pairs in cases of collinearities or plain substitutions.

I have found this system very flexible and efficient, especially in the context of parsing and matching, where the scene (or the current object) is subject to continual change.

This should be enough.

15.0 BIBLIOGRAPHY

AGIN G J
"REPRESENTATION AND DESCRIPTION
OF CURVED OBJECTS"
Stanford AI Memo AIM-173
Stanford University
October 1972

BAUMGART B G
"GEOMED"
Forthcoming AI Memo or Operating Note
Stanford University
1973

BAUMGART B G
"IMAGE CONTOURING AND COMPARING"
Forthcoming AI Memo
Stanford University
1974

BINFORD T O
"A VISUAL PREPROCESSOR"
Internal Report
MIT Project MAC
April 1970

BRICE C R & FENNEMA C L
"SCENE ANALYSIS OF PICTURES USING REGIONS"
AI Technical Note 17
SRI Project 7494
November 1969

CLOWES M B
"ON SEEING THINGS"
AI Journal
Spring 1971

FALK G
"COMPUTER INTERPRETATION OF IMPERFECT LINE-DATA
AS A THREE-DIMENSIONAL SCENE"
Stanford AI Memo AIM-132
Stanford University
August 1970

FELDMAN J A et al.
"THE STANFORD HAND-EYE PROJECT"
Proc. IJCAI (pp. 521-526)
May 1969

GRAPE G R
"COMPUTER VISION THROUGH SEQUENTIAL ABSTRACTIONS"
Internal Memo
Stanford AI Project
June 1969

GRAPE G R
"ON PREDICTING AND VERIFYING MISSING ELEMENTS IN
LINE-DRAWINGS, BASED ON BRIGHTNESS DISCONTINUITY
INFORMATION FROM THEIR INITIAL TV-IMAGES"
Course Project Report for CS225 and CS382
Stanford University
March 1970

GUZMAN A
"COMPUTER RECOGNITION OF THREE-DIMENSIONAL OBJECTS
IN A VISUAL SCENE"
MAC-TR-59
MIT Project MAC
December 1968

HUECKEL M
"AN OPERATOR WHICH LOCATES EDGES
IN DIGITIZED PICTURES"
Journal of the ACM (pp. 113-125)
January 1971

HUECKEL M

"A LOCAL OPERATOR WHICH RECOGNIZES EDGES AND LINES"

Journal of the ACM

To be published 1973

HUFFMAN D A

"LOGICAL ANALYSIS OF PICTURES OF POLYHEDRA"

AI Group, Technical Note No. 6

SRI Project 7494

May 1969

ORBAN R

"REMOVING SHADOWS IN A SCENE"

AI Memo 192

MIT AI Laboratory

August 1970

PINGLE K K & TENENBAUM J M

"AN ACCOMMODATING EDGE FOLLOWER"

Proc. IJCAI

September 1971

ROBERTS L G

"MACHINE PERCEPTION OF THREE-DIMENSIONAL SOLIDS"

Technical Report No. 315

MIT Lincoln Laboratory

May 1963 (reissued May 1965)

SHIRAI Y

"A HETERARCHICAL PROGRAM FOR RECOGNITION OF POLYHEDRA"

AI Memo No. 263

MIT AI Laboratory

June 1972

SOBEL I

"CAMERA MODELS AND MACHINE PERCEPTION"

Stanford AI Memo AIM-121

Stanford University

May 1970

SWINEHART D C & SPROULL R F

"SAIL"

Stanford AI Project Operating Note No. 57.2

Stanford University

January 1971

TENENBAUM J M

"ACCOMMODATION IN COMPUTER VISION"

Stanford AI Memo AIM-134

Stanford University

October 1970

UNDERWOOD S A & COATES C L
"VISUAL LEARNING AND RECOGNITION BY COMPUTER"
Technical Report No. 123
Information Systems Research Laboratory
Electronics Research Center
University of Texas at Austin
April 1972

WALTZ D L
"GENERATING SEMANTIC DESCRIPTIONS FROM
DRAWINGS OF SCENES WITH SHADOWS"
AI TR-271
MIT AI Laboratory
November 1972

WINSTON P H
"LEARNING STRUCTURAL DESCRIPTIONS
FROM EXAMPLES"
MAC-TR-76
MIT Project MAC
September 1970